Vrije Universiteit Brussel

FACULTEIT VAN DE WETENSCHAPPEN
Departement Computer Wetenschappen
Web & Information Systems Engineering

A Conceptual Modelling Approach for Behaviour in Virtual Environments using a Graphical Notation and Generative Design Patterns

Proefschrift ingediend met het oog op het behalen van de graad van Doctor in de Wetenschappen

Bram Pellens

Academiejaar 2006-2007

Promotor: Prof. Dr. Olga De Troyer
Co-Promotor: Dr. Frederic Kleinermann

Samenvatting

Virtuele Realiteit (VR) bestaat al vele jaren en de meeste onder ons zijn er op één of andere manier wel eens een keer mee geconfronteerd geweest. Sinds de introductie van deze innovatieve vorm van interactie tussen mens en computer, in het midden van de jaren 80, heeft Virtuele Realiteit al veel interesse opgewekt en dit zal waarschijnlijk nog vele jaren zo blijven.

Doorgaans gaat Virtuele Realiteit over het creëren van zogenoemde Virtuele Omgevingen, die geloofwaardig, realistisch, of op zijn minst visueel attractief zijn. De allereerste applicaties die gebruik maakten van Virtuele Realiteit, hadden niet echt realistische Virtuele Omgevingen. Anno 2007 zijn de nodige technologiën zodanig gevorderd dat realistische omgevingen binnen de mogelijkheden liggen. Het meeste onderzoek in Virtuele Realiteit werd gedaan op het gebied van het modelleren van het statische gedeelte van de Virtuele Omgeving, namelijk de objecten en hun uiterlijk alsook hoe deze objecten te combineren om zo de Virtuele Omgeving te construeren. Dit resulteerde in een aantal software applicaties en API's, die de taak van de ontwikkelaar aanzienlijk verlichten, vooral wat betreft het modelleren van het statische gedeelte. Virtuele Omgevingen zijn echter zeer dynamisch van nature de objecten hebben gedrag en er is interactie mogelijk tussen de objecten onderling en tussen de gebruikers en de objecten. Tot op heden is er, in contrast met het statische gedeelte, zeer weinig onderzoek verricht naar het modelleren van het dynamische gedeelte van de Virtuele Omgeving. Het gedrag wordt meestal manueel gespecificeerd door middel van een specifieke scripttaal of een algemene programmeertaal. Het gevolg is dat het modelleren of specificeren van gedrag veel tijd kost en niet mogelijk is voor mensen zonder achtergrondkennis over programmeren. Daarbij komt nog dat huidige Virtuele Omgevingen steeds realistischer moeten zijn en dus meer gesofistikeerd gedrag vereisen. Daarom is er nood aan nieuwe technieken om het modelleren van gedrag te vereenvoudigingen en toegankelijk te maken voor een breder publiek. Met andere woorden, er moet een antwoord gezocht worden op de vraag: "Hoe kan de ontwikkeling van Virtuele Omgevingen, en in het bijzonder het modelleren van gedrag, verbeterd worden?

In dit proefschrift wordt in het licht van de bovenstaande vraag, een benadering beschreven voor het modelleren van gedrag. Het werk is verricht in de context van VR-WISE, een reeds bestaande benadering voor het modelleren van Virtuele Realiteit. Het hoofddoel van VR-WISE is het vergemakkelijken en versnellen van de ontwikkeling van Virtuele Omgevingen. Dit wordt in VR-WISE bewerkstelligd door een expliciete conceptuele modelleerfase toe te voegen aan de algehele ontwikkelingsproces. De ontwerper kan een Virtuele Omgevingen specificeren door middel van hoog niveau specificaties en zonder rekening te moeten houden met implementatie details. VR-WISE ondersteunde echter enkel het modelleren van het statische gedeelte van een Virtuele Omgeving. De doelstelling van dit proefschrift was dan ook om VR-WISE uit te breiden zodat ook het dynamische gedeelte (gedrag) van een Virtuele Omgeving kan gemodelleerd worden via hoog niveau specificaties. Eerst zal een conceptuele benadering voor het modelleren van gedrag gentroduceerd worden samen met een grafische modelleertaal. Deze modelleertaal laat de ontwerper toe om gedrag grafisch te specificeren, op een hoog niveau en zonder de noodzaak om kennis over programmeren te hebben. Ten tweede zal deze grafische modelleertaal uitgebreid worden om tegemoet te komen aan de beperkingen van een grafische notatie. Namelijk, wanneer de modellen complex worden, zijn grafische diagrammen vaak niet meer overzichtelijk. Dit gebeurt vooral wanneer we te maken hebben met complex gedrag. Daarom is er een raamwerk ontwikkeld, gebaseerd op het concept van ontwerppatronen, dat toelaat om meer complex gedrag te ontwerpen door het gebruik van gedragspatronen. Minder ervaren ontwerpers kunnen voorgedefinieerde patronen gebruiken, terwijl meer ervaren ontwerpers hun eigen patronen kunnen definiren. Ten slotte beschrijven we in dit proefschrift hoe de benadering gevalideerd werd door middel van een proof-of-concept software applicatie die de benadering ondersteunt. Ook zal er gerapporteerd worden over een experiment waarin leken deze tool gebruikt hebben om gedrag te modelleren. Bovendien zijn er ook enkele voorbeelden uitgewerkt en beschreven in dit proefschrift.

Abstract

Virtual Reality (VR) has been around for many years and most of us have, in some way or the other, been confronted with it, at least once. Since the introduction of this innovative form of human-computer interaction in the mid nineteen eighties, VR has gained a lot of interest and will continue to do so for many more years.

Basically, VR is about creating Virtual Environments (VEs), which are believable, realistic, or at least visually attractive. However, this was not the case for the very first VR applications. Over the last ten years, the technology has evolved considerably and fairly realistic systems can be built nowadays. Most of the research efforts have been put into improving the modelling of the static part of a VE, i.e. the visual appearance of objects and how to compose the overall scene. As a result, a number of software tools and application programming interfaces have been developed. These packages have eased the task of the designer mainly on the modelling of the static part of a VE. However, in essence VEs are really dynamic, i.e. objects have behaviours and users can interact with these objects. Unfortunately, the modelling of the dynamic part of a VE did not receive the same attention and therefore has not undergone the same progress as the static part. The specification of behaviour is mostly being done at a low level by manually programming it using either dedicated scripting languages or general purpose programming languages. As a consequence, the process of building behaviours is still not accessible to designers without programming skills. In addition, current VR applications are becoming more realistic and therefore require more complex behaviours. For this reason, there is a strong need to have new ways to make the modelling of behaviour easier and more intuitive. In other words, there is still no adequate answer to the question "How can the development process of VR applications, and more precisely the aspect of modelling the behaviour, be improved further?"

This dissertation presents an approach that is addressing the above question. The work is done in the context of VR-WISE, an existing modelling approach for Virtual Reality. The main goal of the VR-WISE approach is to ease and shorten the development of VEs by including an explicit conceptual modelling phase into the overall design process. It enables the designer to specify a VE through intuitive high-level specifications without the need of considering any implementation details. Until now, VR-WISE only focused on modelling the static part (i.e. scene) of a VE. The purpose of this dissertation has been to extend VR-WISE in order to facilitate the modelling of the dynamic part (i.e. behaviour) of a VE. Firstly, this dissertation will introduce a behaviour modelling approach and an associated graphical *Behaviour Modelling Language*. This language allows designers to visually specify behaviour at a high level and without requiring advanced programming skills. Secondly, the dissertation will describe how the behaviour modelling approach has been extended to overcome some of the limitations associated with graphical notations when it comes to modelling complex behaviours. A *Behavioural Design Patterns Framework* has been developed to facilitate the specification of more complex behaviours by incorporating behavioural patterns into the graphical modelling language. This framework allows less experienced designers to use predefined behavioural patterns while more experienced designers can create their own behavioural patterns. Thirdly, the dissertation will explain how this approach has been validated by implementing a proof-of-concept software tool supporting VE development using the VR-WISE approach. It will also report on an experiment performed showing how laymen have used this tool to model behaviours for real case scenarios. Furthermore, a number of other case scenarios have been elaborated and are presented in this dissertation.

Acknowledgements

The period in which the work for this dissertation has been done, was testing, but in the first place exciting, instructive, and fun. Without the help, support, and encouragement from several persons, I would never have been able to finish this work.

First and foremost, I would like to express my deep and sincere gratitude to my promotor, Prof. Dr. Olga De Troyer, Head of the Web & Information Systems Engineering (WISE) Laboratory of the Vrije Universiteit Brussel. Her wide knowledge and her logical way of thinking have been of great value for me. Her understanding, encouraging and personal guidance have provided a good basis for this dissertation.

I am also deeply indebted to Dr. Frederic Kleinermann, my co-promoter. The many conversations I had with him on various topics gave me valuable insights in the domain of Virtual Reality. It will also be very hard for me to repay him for the countless hours he spent on proofreading and discussing my text. I hope we can continue to cooperate in the future.

Next, I would like to thank the members of my jury, Prof. Dr. Marc Cavazza, Prof. Dr. Nicholas Avis, Prof. Dr. Ir. Geert-Jan Houben, and Prof. Dr. Viviane Jonckers, first of all to be in my jury and secondly for providing me with good pointers to further improve the quality of my text.

Thanks also to all my colleagues and ex-colleagues at the Department of Computer Science for providing a good working atmosphere, especially the persons in the Web & Information Systems Engineering (WISE) Laboratory, namely Wesley Bille, Sven Casteleyn, Peter Plessers, Raül Romero, Haithem Mansouri and Abdelgani Mushtaha. They started as colleagues, but became my friends.

Obviously, I would not be sitting in front of my computer typing these acknowledgements without my parents. I owe my parents, Diane and Jacky Pellens much of what I have become. I thank them for their love, their support, and their confidence throughout the past twenty-six years. My parents have always put education as a priority in my life. They have done everything in their power to make this possible. They raised me to set high goals for myself. I always needed to work hard to achieve my goals in life and they have always been there for me as an unwavering support. I dedicate this work to them.

I owe my loving thanks to my girlfriend Els Klerkx for her love and patience during the PhD period. She supported me with an endless number of things which allowed me to focus on nothing but my work. She practically put her life on hold for me. Nonetheless, she had to endure many of my mood swings. I thank her for being there for me in times when I needed her most.

My special gratitude is also due to my sister Ine Pellens and to the rest of my family for their loving support.

Last, but not least, special thanks go to my best friends whom I have been neglecting during the time I was writing up my dissertation: Kevin Kuppens, Bert Creylman, Larissa Mandos, Tine De Wit, Catherine Cober, Andy Kellens, Serge Kovacs, Roel Peeten, Tom Peeten, Annika De Graaf, Wendy Coene and off course all the people of the VUB Diving Center.

Finally, I would like to thank all the people I unwittingly forgot to mention here. Many people contributed in some way to this PhD and I thank all of them.

Please note that the work reported in this dissertation has been financially supported by the Institute for the Promotion of Innovation by Science and Technology in Flanders (IWT) through the VR-DeMo (IWT 030248) project, a research project in cooperation with Expertise Centrum Digital Media (EDM - Universiteit Hasselt). Therefore, I also thank the IWT for this support and the colleagues at EDM for the four years of cooperation.

Contents

List of Figures

List of Tables

Glossary

Actor A representation of an object (Concept/Instance) that is involved in a behaviour.

Animation A predefined time sequence of visual changes in a 3D model or Virtual Environment. These changes can be anything from translations and rotations up to changes in size, colour, texture and so on.

API Application Programming Interface.

Behaviour Definition Diagram (BDD) A diagram defining the behaviour of a Virtual Environment separated from the objects and the interactions.

Behaviour Invocation Diagram (BID) A diagram specifying the connections between the behaviours, the objects and the interactions.

Bounding Box A rectangular bounding volume that completely contains an object.

Concept A description of a set of objects having similar characteristics.

Conceptual Model A description of a portion of the 'real world' that is of interest in a particular application domain.

Domain Expert A person with special knowledge or skills in a particular area, being the domain for which the VR application needs to be built.

Generative Design Pattern A design pattern with the additional ability to generate program code from it.

Instance An individual representatitive of a concept.

Layman A person who is a non-expert in the field of Virtual Reality.

Object An actual 3D model as a representative of an instance.

Ontology An ontology comprises the definitions of the concepts, individuals, properties and relations which makes up a conceptualization.

Reference Frame A particular perspective employed by the designer to describe and/or observe a motion effectively.

Scenario An outline or model of an expected or supposed sequence of events happening in the Virtual Environment

Scene Graph A directed acyclic graph containing the spatial representation of a Virtual Environment.

Simulation An attempt to model a real-life situation through a Virtual Environment and in which the output (behaviour) differs according to the set of initial parameters assumed for the environment.

Structure Chunk (SC) A diagram defining the composition of a set of objects at a particular instant.

Virtual Environment (VE) A computer-generated, and computer-maintained, three-dimensional environment, which can be navigated and interacted with by its users.

Virtual Reality (VR) The combination of technologies to create Virtual Environments as well as those that are involved in enabling them.

Virtual World See Virtual Environment.

CHAPTER 1

Introduction

In 1965, Ivan Sutherland introduced for the first time the notions of a new form of interaction between humans and computers. He described it as follows [Sutherland, 1965]: "The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be good enough to sit in. Handcuffs displayed in such a room would be confining, and a bullet displayed in such a room would be fatal". If this notion about so-called Virtual Environments would become true, it would drastically change the field of computer graphics forever. However, every new technology brings not only new possibilities but also many new problems. On of these problems, and still one of the most challenging aspects of the design of a Virtual Environment is the design of the behavioural aspect. This will be the topic of this dissertation.

1.1 Research Context

1.1.1 Virtual Reality

In 1963, Ivan Sutherland came up with a revolutionary computer application, called *Sketchpad*, which is generally accepted as being the origin of modern computer graphics. The system was one of the first ever that was designed to use a Graphical User Interface (GUI). It allowed users to interactively (something which was very rare at that time) draw on a screen using a *light pen* (a new kind of pointing device in analogy with a normal pen that is used to draw on paper). Many of the ideas behind the modelling tools used today (e.g., from simple recreational drawing tools up to professional CAD tools), were initially developed for this system. There is for example the rubber-banding technique to draw lines on the display. The most important achievement was the concept of *graphical object* which was introduced as an entity with its own semantics and behaviour. Furthermore, there was the idea of having a master object that could be instantiated into many duplicates. If the user changed the master drawing all the instances would change as well. Although the system was quite primitive in its first implementation, it was the starting point of a new era in computer graphics.

Once there was this capability of interactively building a collection of graphical objects (possibly interacting with each other), there was an urge for the possibility of composing these objects into a larger whole resulting in a new *phantom world*. This world is actually a set of objects maintained by the computer, following the laws and behaviours they were programmed to. As a consequence, a kind of illusion of the real world can be maintained by the computer. The user can interact with this world and can have the sensation of 'being part' of it.

This has lead to the notion of Virtual Reality as it is still known today. However, it was not until 1986 that Jaron Lanier officially coined the term. Also the terms *Virtual Environments* and *Synthetic Environments* have emerged amongst others. All these terms are used interchangeably in the community but Virtual Reality and Virtual Environments are by far the most popular ones and the most used. A nice description, about what Virtual Reality is all about, is given in [Vince, 2004]:

> *Virtual Reality is about creating acceptable substitutes for real objects or environments, and it is not really about constructing imaginary worlds that are indistinguishable from the real world.*

In the remainder of this work, *Virtual Reality* (VR) will be referred to as the broad term to denote not only the systems that are used in the process of modelling these substitutes but also all the technologies that are involved to make them happen. *Virtual Environments* (VEs) or *Virtual Worlds* (VWs) will be used to refer to the computer-generated (and computer-maintained) three-dimensional environments, which can be navigated and interacted with by the users. They are often the result of a Virtual Reality system.

The fact that Virtual Environments provide substitutes for the real physical environments together with the ability of the user being part of these Virtual Environments, gives their use major benefits:

- When comparing it to the physical environment that would otherwise be used, no or **very little space** is required for the complete system. Depending on the type of system, it can take the space of a single pc with a screen up to a small room.

- It has the ability to represent large amounts of information in a more intuitive way than with 2D visualizations with a very **high accuracy** and **realism**. The Virtual Environment is also much cheaper than if the real physical models would have been built.

- The content of a Virtual Environment is no longer a snapshot of the information at a particular time but the evolution of the data can be shown and consequently true **animations** and realistic **simulations** can be made.

- The user can **navigate** inside the Virtual Environment instead of just being an observer. He can move around and explore all the features of the 3D scene in different ways like walking from one place to another, but he can also fly and jump. This was not possible with traditional CAD[1]-based systems.

- Not only Virtual Environments can be navigated, but they can also be **interacted** with. Many new ways of interaction, beyond the standard keyboard and mouse input and screen output, have been developed for a user to enable manipulation of the data visualized by the Virtual Environment.

- The same Virtual World can be **shared** by many users at the same time. In this way, the people can interact with each other or collaborate on a particular task in a natural way without actually being in the same physical location.

People really start to see the benefits that Virtual Reality has to offer to a limitless range of potential application areas [Magnenat-Thalmann and Thalmann, 2000].

First of all, Virtual Reality has many applications in the industrial world. An advantage is that virtual prototypes of products can be created. The prototypes not only can be visualized, but they can also be examined and evaluated before the product is actually built. Working with the virtual prototype instead of with a real prototype greatly speeds up the design process and potentially reduces the development costs. Virtual Reality is often used as a means of visualizing data as well. In the architectural domain for example, complete building projects could be virtually constructed and experiments can be made with different lighting conditions, space layouts and so on. Also in the scientific visualization domain, Virtual Reality is useful in order to display large amounts of scientific data where the data can be interpreted more easily. Virtual Reality is also used as a means for remote operation or teleoperation. People working in dangerous environments that make it difficult or even impossible to perform their tasks, can now be dislocated to operate in more safe Virtual Environments and still maintain full control over these environments. Virtual Reality is a promising technology in education and training. The most famous example, and also one of the first applications to use Virtual Reality, is the flight simulator where future pilots are safely taught how to fly a plane in various circumstances. Other

[1]Computer Aided Design

examples are surgical simulators, driving simulators, and so on. Not only simulators but also other types of applications such as virtual museums[2] are attributed to this sector. Last but not least, there is the entertainment sector that is a fast growing industry and a big business. The computer games produced nowadays are more and more realistic. It even becomes harder to make a distinction between a Game World and a Virtual World. Virtual Reality could play a central role in the next generation of computer games. Vice versa, in the serious gaming community, game technology and game design principles could are also applied nowadays for a primary purpose other than pure entertainment[3].

The list of applications given here is far from complete and many more applications can be built using Virtual Reality. Only a few were listed to give the reader an idea of what are the possibilities of this technology, which is known to many people but at the same time still unknown to many others. Nonetheless, it is without doubt that Virtual Reality has already attracted a lot of interest and will continue to do so in the near future.

In the last decade, a lot of efforts have been spent to bring 3D and Virtual Reality to the World Wide Web (WWW). Initially, the WWW was only capable of communicating text and 2D graphics through its HyperText Markup Language (HTML). It did not have any support for the third dimension and so a new language was required which would enable the distribution of dynamic, interactive 3D worlds. The Virtual Reality Modeling Language (VRML) was created during the mid '90s to address this issue.

Like every technology, VRML has improved over time. Nowadays VRML is known under the name of eXtensible 3D (X3D) and is maintained by the Web3D[4] consortium group. X3D (VRML) has matured up to a point that it enables full 3D applications on the Web. The combination of the Internet and Virtual Reality provides a powerful medium for even a wider range of application areas than those discussed above.

The popularity of Web3D enabling technologies is continuously growing due to both an increasing demand for interaction between online users and an expanding interest in Web-based applications overall. The emergence of so-called Web-based Virtual Environments [Hughes et al., 2002] are a perfect example of such applications and their successes prove the fact that Virtual Reality on the Web is growing in popularity. For example, Second Life[5] created by the Linden Lab, is the most famous three-dimensional

[2]http://virtueelmuseum.vub.ac.be
[3]http://www.seriousgames.org
[4]http://www.web3d.org
[5]http://www.secondlife.com

Virtual World[6]. The Active Worlds[7] and There[8] systems are similar 3D Virtual Reality platforms hosted by Activeworlds and Makena Technologies respectively. Users in these online Virtual Environments can explore existing environments, build new ones, do online shopping, socialize with other users, play games and so on.

Having introduced the origin of VR and its implications in different domains, this section will explain the different components that compose a Virtual Environment in more details.

1.1.2 Anatomy of a Virtual Environment

A complete Virtual Reality system usually consists of both hardware and software. The hardware components are computers, input devices (e.g., 3D mouse, gloves,...), output devices (e.g., haptic devices, audio,...), displays (e.g., head-mounted displays, CAVE,...). We will not go into further details on these components since this is outside the scope of this dissertation. The interested reader is referred to [Stuart, 2001] for an elaborated description of these issues and other technologies involved. However, some extra attention will be given to the Virtual Environment itself, generated by the software, as this will be our main area of interest. A Virtual Environment can basically be broken down into three parts: the static scene, behaviour and interaction [Kessler, 2002].

1.1.2.1 Static Scene

One of the major parts of a Virtual Environment is the static scene, i.e. the scene with its objects as displayed by the computer. The scene encapsulated the complete Virtual World with its virtual objects. To be displayed on a screen, the world and the objects have some geometry attributes, as well as some material attributes. The geometry represents the shape of the object. It is usually defined by atomic primitives such as points, lines, or polygons or more complex constructs such as spheres, boxes, cylinders, cones, and so on. The material is used to tell the computer how the shape eventually should look like in the scene and is given in terms of colours and textures. All the objects also have a position and an orientation in space (defined in a three-dimensional coordinate system). There may be more attributes that can be related to an object but not all of them will be discussed here.

[6]On August 30, 2007, around 1.6 million users were reported to have logged-in over the last 60 days. It has its own economy and currency (http://secondlife.com, access date September 3, 2007)

[7]http://www.activeworlds.com

[8]http://www.there.com

1.1.2.2 Behaviour

An equally important aspect of a Virtual Environment as the scene is the behaviour, i.e. how the world and its objects act or react. Every object can have a set of behaviours that can be executed depending on the context the object is in. These behaviours can be divided into two categories:

- *Environment-independent* behaviours do not need external stimuli from the environment in order to be executed. They can be either time-dependent, where an object's properties are changed over time or at a certain point in time, or time-independent, where a fixed sequence of changes (that may be time-dependent) are specified but the overall behaviour is not depending on the time.

- The *environment-dependent* behaviours are performed as a reaction to particular external stimuli. They can be either event-driven or constraint-driven. In the case of event-driven, an event is fired when the user performs actions like interacting or when another object performs an action like collision. For constraint-driven, a change is made to an object on the basis of its relationships with other objects in the scene.

Different levels of behaviour can be distinguished as described in [Roehl, 1995]. There are basically four levels in a hierarchical order. The first level contains very simplistic behaviours which are directly modifying an entities' attributes. The second level is about changing these attributes of the entity over a larger range of time. These behaviours are composed to create level-three behaviours and thus consist of the simpler actions from the second level. Finally, the fourth level is then all about selecting which behaviour should be executing and involves decision-making. This dissertation is mostly concerned with the behaviours of the third level and in a smaller degree with the behaviours of the fourth level.

1.1.2.3 Interaction

An aspect closely related to the behaviour is interaction since interaction is often triggering behaviour. There are basically two types of interaction. Firstly, *user-interaction*, which is defined as every action taken by the user with the intention to either change the environment, or to perceive the world. In a Virtual Environment, interaction is composed of different basic interaction tasks. The basic interaction tasks can mainly be divided into three categories: navigation, selection and manipulation. For each of the basic interaction tasks different interaction techniques are available. Navigation refers to the user interactively positioning and orientating himself in the VE using techniques like walking, or flying. Selection refers to the picking of one or more objects for some purpose by means of for instance laser pointing, or cone casting. Manipulation refers to the positioning and orientating of objects using techniques such as object-in-hand or voodoo dolls. Secondly, *object-interaction* is defined as the action that one virtual object exerts on another. Usually, this is reflected as a kind of collision.

Since the interaction aspect is not the main topic of this dissertation, it will not be discussed in more detail. The interested reader is referred to [Bowman, 1999] for an elaborate description of interaction techniques in Virtual Environments.

1.2 Problem Statement

Virtual Reality is starting to hold its promises that were made in the late '80s. The technology (in terms of hardware at least) has matured a lot and from a technological point of view, fairly realistic Virtual Worlds can be built. Although the technology has evolved quickly over the last ten years, this has not been the case for the development of Virtual Reality applications. It is still not easy to develop Virtual Reality applications. The modelling of behaviour is difficult in comparison with the modelling of the static scene (i.e. the visual appearance of objects and their position inside the Virtual World). This is partly due to the fact that most of the improvements have been made towards modelling the visual appearance of objects populating the Virtual World and composing the static scene more easily. Over the last twenty years, these improvements revolved around the graphics packages, authoring tools, APIs, and file formats. A short history will be given in the next sub-section.

However, the modelling of behaviours did not receive the same kind of attention until recently and it is still nowadays not easy, especially for people not experienced in Virtual Reality, to model behaviours. In the last six years, the game industry has provided a way of allowing users to customize their game [Busby et al., 2004]. However, the way the behaviours are modelled are mostly through dedicated scripting languages which is not intuitive for non-VR-experts often having no adequate programming skills and even worst, these scripting languages are also changing from one game platform to the other one. As a result, the complete language has to be relearned over and over again. Furthermore, people are also more demanding today in terms of behaviour than ten years ago. This is due to games becoming more and more realistic. As VEs are also becoming more accessible to a larger audience thanks to the Internet, it is therefore necessary to develop approaches and tools which can ease the task of the designer when modelling behaviours. This will be the aim of the dissertation. The work presented in this dissertation does not provide a complete solution for the problems involved with the development of Virtual Reality applications. Nevertheless, some important contributions in the domain of the development of the behavioural part of a Virtual Reality application are made, which allow making the development more accessible to a larger public.

This section will be split into two sub-sections. The first sub-section will give a short history on the improvements which have been made for modelling the visual appearance of objects as well as composing the static scene of a VE. The second sub-section will explain why modelling the behaviour of a VE is difficult and how this can be changed.

1.2.1 Modelling the Static Scene

Figure 1.1 gives an overview of the different application layers involved in Virtual Reality development.

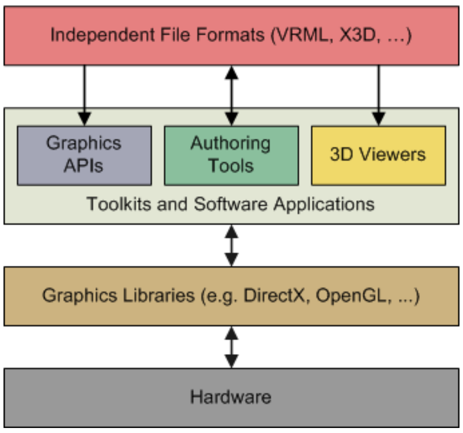

Independent File Formats (VRML, X3D, ...)
Graphics APIs
Authoring Tools
3D Viewers
Toolkits and Software Applications
Graphics Libraries (e.g. DirectX, OpenGL, ...)
Hardware

Figure 1.1: *Virtual Environment application stack*

1.2.1.1 Graphics packages

The lowest level, on which all the software for designing VR applications are based, are the graphics packages. Graphics packages provide an interface between the software and the hardware for creating and manipulating graphics and pictures. Most of the time, the graphics packages provide users with a variety of functions for creating graphics and properly render them on the screen as well as functions for transforming these graphics. They also provide some housekeeping tasks such as clearing a screen display, initializing parameters and so on.

In terms of graphics packages, international and national standards-planning organizations did cooperate in an effort to develop a generally accepted standard for computer graphics. This work led in 1984 to the Graphical Kernel System (GKS). It was followed by another library called Programmer's Hierarchical Interactive Graphics Standards (PHIGS). At the time when these packages were developed, the graphics workstations from Silicon Graphics, Inc (SGI[9]) became increasingly popular. These workstations came with their own library of routines for graphics called Graphics Library (GL). These routines were designed to make real-time rendering possible. Later, the GL was extended to cope with other hardware systems in order to be more hardware-independent. This has given birth to the currently known graphics library called OpenGL[10]. In the middle of the '90s, another graphics library, called DirectX[11], was created by Microsoft as a competitor to OpenGL. Today, both OpenGL and DirectX are used on PCs and mobile devices for creating and manipulating graphics and pictures. Most of the software for designing Virtual Reality applications nowadays is based on these two graphics libraries.

To use one of these graphics libraries, one needs knowledge about programming as well as mathematics. These libraries are therefore not at all intuitive to use for non-VR-experts. They usually consist of a wide collection of commands, with long and unintuitive names, reducing even more the usability. Furthermore, it requires time to learn how to program the graphics pipeline using the libraries. Today, this situation is even worse as the graphics pipeline[12] can be programmed to have all kinds of visual effects. Moreover, these graphics libraries are constantly changing in order to incorporate new features.

1.2.1.2 Scene Graph and Complex Scenes

To model objects in order to be displayed on a screen, they firstly need to be represented by a geometry (vertices), a topology (relationships between the edges and faces) and material properties. Then, the objects can be manipulated in various ways such as translated, rotated, scaled, and various combinations of these operations. These operations (or transformations) can be accomplished by matrix transformations provided by graphics libraries like OpenGL or DirectX. However, objects do not exist in isolation, but instead are often interdependent, i.e. a transformation applied to one object (or part of an object) often propagates to other objects. As an example, consider a car having a body and four wheels. As the car is positioned to another position, the four wheels and the body also need to be repositioned. To program such operations, routines from the graphics libraries mentioned above can be used. However, from a software development point of view, this will not be efficient and intuitive as the same routine needs to be called several times and each part of the car is modelled separately. For instance,

[9]http://www.sgi.com/

[10]http://www.opengl.org/

[11]http://www.microsoft.com/directx/

[12]The different stages that need to be done in order for a representation of a three-dimensional scene to be converted in a 2D image on the screen.

each wheel of the car has the same geometry and material, only the position will be different. Therefore, only the matrix transformation of each wheel must be stored. To optimize the design process of modelling such complex objects and scenes, a data structure was introduced. This data structure is known as *scene graph*.

In general terms, a scene graph is a special type of data structure used to describe scenes, i.e. the objects that are contained in it together with their characteristics. More technically, a scene graph is typically tree data structure. Every node can have a number of child nodes but only one parent node. At each node, there is a corresponding transformation matrix specifying how the object at that node is related to its parent node. If an operation is performed to a node, it is reflected to all of its child nodes, but not to its parent node. An operation can hence be described by a single transformation matrix, which is formed by the multiplication of the matrices on the path connecting the nodes. Having these characteristics, related objects can now be grouped into a compound object that can then be moved (transformed or selected) as easy as a single object. However, a scene graph can also be a *directed acyclic graph* (DAG) which is similar to a tree, except that in the DAG branches are allowed to merge. That is, different branches might refer to the same leaf node as its child node.

Therefore, the scene graph can be seen as a kind of abstraction mechanism as it provides a way to model complex scenes and its objects in a more efficient way than with the low-level graphics libraries. Furthermore, it is also more intuitive for a non-VR-expert to think in terms of a graph, like for instance in the example of the car. At this moment, the scene graph approach is still the most widely used modelling paradigm in Virtual Reality.

As the low-level graphics packages do not provide such a structure, a number of APIs have been built on top of the low-level graphics packages and incorporate the concept of scene graph. Some examples are Java3D[13], Performer[14], and OpenSceneGraph[15]. Recently, Microsoft released the first version of XNA[16], an interesting set of tools to facilitate 3D computer game design. With the XNA Game Studio Express toolkit, developers can create games, for either the XBOX or a normal PC, using the XNA Framework. There is one difference though between XNA and other toolkits. While developers have always been able to make games, it is claimed that the XNA has reduced the steep learning curve for game development and made it into something more attainable for developers such as students and hobbyists. In practice, the XNA is nothing more than an abstraction layer on top of the DirectX graphics library and thus one still needs to have programming skills and preferably some background on DirectX, in order to really start

[13]http://www.java3d.org/
[14]http://www.sgi.com/products/software/performer/
[15]http://www.openscenegraph.com/index.php
[16]http://msdn.microsoft.com/directx/XNA/default.aspx

working with it.

All these APIs also provide a way to specify behaviour and interactions. Furthermore, they provide a number of additional routines that facilitate the programming part for behaviour and interaction. However, the bulk of the work is still very much oriented towards the languages in which the APIs are programmed.

1.2.1.3 Platform Independence and File Formats

As the Internet is becoming part of our life nowadays, it is natural that computer graphics applications and therefore Virtual Reality applications are also appearing on the Web. Good examples of this are the 3D online communities that were mentioned before. In the mid '90s, the Virtual Reality Modeling Language (VRML) was developed. VRML is an independent file format to describe the 3D content in a declarative way. A specification of a Virtual Environment in VRML needs to be interpreted by a suitable Web browser having an appropriate player to parse the files in order to construct the scene. At the end of the '90s, it became clear that VRML was too limited and needed to be extended. At the same time, the XML file format was well accepted and used by a number of industrial companies developing Internet applications. For these reasons, the consortium group Web3D in charge of extending VRML, decided to change the VRML into an XML-like format. The VRML has been renamed X3D for eXtentensible 3D (X3D) [Walsh and Bourges-Sevenier, 2000]. Both VRML and X3D support the concept of scene graph and the notion of *interpolators* to specify simple animations and *scripts* for modelling more complex behaviour together with *sensors* for modelling the interaction and *routes* for connecting the elements together. One of the main differences between X3D and VRML is the definition of an XML encoding for the language. This allows for easier integration of 3D content with other Web content, technologies, and tools. In 2004, it became an ISO/IEC standard and now it can be seen as the de facto standard for 3D on the Web.

At the end of the '90s and also to enable 3D on the Web, SUN[17] introduced the API called Java3D [Selman, 2002]. Unlike VRML, Java3D is an extension on the Java programming language. The idea is to use Java as a platform independent framework and to build a set of routines to create and to manipulate graphics applications on top of the two mostly used graphics libraries namely OpenGL and DirectX to perform native rendering. The 3D scene specification, application logic and scene interactions are programmed in an object-oriented way using the Java language. However, Java3D facilitates this by supporting the concept of scene graph and new routines to model behaviour and interaction as well as to incorporate different types of

[17]http://www.sun.com

devices. It provides developers with a set of APIs to write either standalone 3D applications for the desktop or Web-based 3D applets that need to be compiled into bytecode and subsequently executed by the Java Virtual Machine (JVM) and brought to the Web via Webstart. The advantage of using Java3D is its platform independence, which is very important in a heterogeneous Internet setting. In that way, the developer can create an application that can run on any machine with any graphical hardware.

1.2.1.4 Authoring Tools

During the last ten years, a number of so-called authoring tools for VR have emerged. An authoring tool is a software package used by designers to easily compose their scenes. For most authoring tools, the basic setup is similar, i.e. 4 different views (3 orthogonal views and one perspective view) on the Virtual World. The three views show the top view, the front view and a side view of the scene. These views are needed simultaneously in order to exactly position the objects. The perspective view is used to show both the object as a solid form and in the correct perspective. The modelling itself is mostly done by visually dragging and dropping elements (objects) of the scene from a toolbox onto the canvas and afterwards editing their properties.

They also allow modelling simple behaviours and interactions and attach them to the objects. They also have strong assistance capabilities. The designer is instantaneously notified of his mistake, so that it can be corrected quickly.

Again these tools are built on the top of the existing graphics libraries and internally work with their own scene graph. In general, one can say that there is less technical knowledge required to master them compared to the low-level programming APIs. In choosing the right authoring tool, a trade-off has to be made between ease-of-use and functionality. The low-end authoring tools usually require little learning but they are restricted in their functionality, while the more high-end authoring tools generally require several weeks to several months to learn how to use them efficiently, but they offer much more functionality and provide more flexibility to the developer. Furthermore, most of the authoring tools have the ability to work with other software packages or standard file formats in the form of importers and exporters (for languages such as VRML and X3D).

1.2.2 Modelling Behaviour and its Complexity

Thanks to the progress made in these authoring tools and high-level graphics APIs described above, it is nowadays possible to model more quickly very realistic Virtual Environments. The combination of these software packages with the power provided by the latest graphics hardware has accomplished this.

These software packages are good for modelling graphics and static Virtual Environments, but Virtual Environments are dynamic in nature, i.e. they include behaviour and support interaction. However, most of the software packages lack high-level support for modelling behaviour and interactions. As the focus of this PhD is on behaviour, we will now explain why the modelling of behaviour is still a very difficult task in comparison to the modelling of a static scene.

Let us first have a look to APIs, like Java3D, and file formats, such as VRML and X3D, to understand why modelling of behaviour is even not easy for computer specialists. The following code shows an example in X3D. It is a standard example coming from educational books originally written for Computer Science students. It is a scene, called EarthSystem, containing two objects: the World (which is a sphere) and a Satellite (which is text). If the user clicks on the Satellite, the whole EarthSystem will make one complete rotation through a smooth ease-in/ease-out (approximated) animation.

```
<Scene>
  <TimeSensor DEF="Timer" enabled="true" cycleInterval="12"
    loop="false"/>
  <OrientationInterpolator DEF="Spinner"
    key="0.00 0.10 0.20 0.30 0.40 0.50
         0.60 0.70 0.80 0.90 1.00"
    keyValue="0 1 0 0, 0 1 0 0.154, 0 1 0 0.6, 0 1 0 1.295, 0 1 0 2.171,
              0 1 0 3.142, 0 1 0 4.112, 0 1 0 4.988, 0 1 0 5.683,
              0 1 0 6.129, 0 1 0 6.283"/>
  <ROUTE fromField="fraction_changed" fromNode="Timer"
         toField="set_fraction" toNode="Spinner"/>
  <Transform DEF="EarthSystem">
    <ROUTE fromField="value_changed" fromNode="Spinner"
           toField="set_rotation" toNode="EarthSystem"/>
    <Group DEF="World">
      <Shape>
        <Appearance>
          <ImageTexture url="earth-topo.gif"/>
        </Appearance>
        <Sphere radius="1.5"/>
      </Shape>
    </Group>
    <Transform DEF="Satellite" rotation="1 0 0 0.3"
      translation="0 0 5">
      <Shape>
        <Appearance>
          <Material diffuseColor="0.9 0.1 0.1"/>
        </Appearance>
        <Text string="Hello X3D Authors !!">
          <FontStyle size="1"/>
        </Text>
      </Shape>
      <TouchSensor DEF="Trigger" enabled="true"/>
      <ROUTE fromField="touchTime" fromNode="Trigger"
             toField="startTime" toNode="Timer"/>
    </Transform>
  </Transform>
</Scene>
```

As it can be seen, making this simple example dynamic with performing just one action is quite laborious. There are two different sensors (a TimeSensor and a TouchSensor) in combination with an interpolator (an OrientationInterpolator) and all these are linked by means of three ROUTE statements.

It is clear that even simple programs (without scripts) are already quite verbose. This gets even worse when dealing with larger scenes, hence building large X3D (or any other specification language for that matter) scenes is not obvious, especially not for non-VR-experts.

APIs like Java3D provide some built-in routines to model behaviour. These APIs need to be programmed and extended in order to model complex behaviour. For this reason, they are only accessible to programmers who also need to have some notions of computer graphics, scene graph and mathematics to model efficiently behaviours.

Considering the authoring tools, one can see that the modelling of the behaviour is done in two different ways. If it concerns simple, well-known behaviour, it can be selected from menus in the GUI and parameterized. If it is a complex behaviour, then it must be modelled using scripting languages. This results in not being accessible to people having no or little knowledge in programming and it still requires users to learn the scripting languages. Furthermore, the scripting languages differ from one authoring tool to another. The traditional and widely used technique to represent object behaviour is called keyframe animation. The general idea of keyframe animation is that at particular key moments in time, a snapshot of the object, together with all its properties, is taken. Such a snapshot is called a key frame. This procedure can be repeated for as many times as needed. The system is then able to calculate all the frames in between by means of an interpolation scheme, using the key frames as a rough sketch of the animation. The keyframe animation technique, which is in most cases directly supported by the design tools, is very powerful. However, it requires a number of actions that are not always easy for the designer [Terra and Metoyer, 2004].

Based on the above observations, one can say that the current practice for modelling behaviours is not very intuitive and obviously it requires the necessary knowledge. As a result, the design tools are only used to create the static scene of the VE which is afterwards imported in a toolkit where the dynamical part is added by means of a conventional programming language

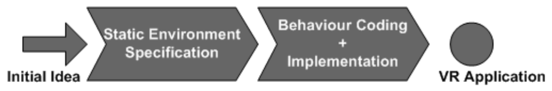

Initial Idea
Static Environment Specification
Behaviour Coding + Implementation
VR Application

Figure 1.2: *The Virtual Reality design process as it is currently perceived*

or a dedicated scripting language such as EcmaScript [Ecma, 2000]. This leaves the task of managing the dynamics of the Virtual Environment to the (Virtual Reality) programmer. In other words, since it is programmed, the design of the behaviour is usually integrated with the implementation of the system.

The current situation in Virtual Environment design is depicted in figure 1.2. The design tools that are available today allow the user to easily build realistic static Virtual Environments without needing a lot of background knowledge. However, this is not the case when it comes to dynamic Virtual Environments. The fact that only the visual part is modelled properly and that the dynamical part is integrated into the implementation phase is mainly due to the history as it was described above. It would be very interesting if the dynamics could also be modelled separated from the actual implementation phase so that this too would become platform (or language) independent. It would be even more interesting if it could be modelled visually as it is nowadays the case for modelling the static scene. Although a lot of research has already been performed in this field (see chapter 2), there are still a lot of improvements that can be made and this will be the main area of investigation in this dissertation.

1.3 Aims and Objectives

As explained above, Virtual Reality is a promising technology. Nevertheless, there is a downside to this as well; the development of a Virtual Environment is still a very difficult, specialized and time-intensive task. One of the reasons is the simplistic development process used. For the technology to become more widely used, it is necessary to make the development of VR applications easier. One way to accomplish this is by extending the development process with a design phase as in classical Software Engineering processes. Such a design phase would allow abstracting from implementation details and concentrate on design issues. The output of the design phase could then be used as input for the implementation phase, and to a certain extend the implementation can be automatically derived. As already stated in this dissertation, we only focus on the behavioural aspects of a Virtual Reality application. Therefore, the ultimate aim of the work presented in this dissertation can be stated as follows:

To facilitate and shorten the development process of Virtual

Environment behaviour by means of high-level specifications and to enable automatic code generation from these specifications.

The situation that we aim for in this dissertation is depicted in figure 1.3. The behaviour modelling is now separated from the actual implementation and made explicit into a proper design phase. In order to achieve the main goal, a number of specific research objectives have been identified:

- Develop a modelling approach for behaviour that allows domain experts to be more involved in the modelling process of behaviour.
- Define a set of modelling concepts that allows describing the behaviour of objects in a Virtual Environment at a high-level. These concepts need to be expressive enough in order to cope with enough complexity.
- Build a framework that allows the designers to easily (re)use existing behaviours and algorithms in the high-level behaviour specifications.
- Provide the necessary tool support for the specification, at a high-level, of the behaviour in a Virtual Environment.
- Enable the automatic translation of the high-level specifications towards the low-level implementation and to provide the necessary support for viewing (loading) the end result.
- Perform user experiments in order to show whether or not this approach really facilitates the development process of behaviour.

As explained before, Virtual Environments are usually built by a number of people, each of them specialized in a certain aspect (e.g., graphics designer, behaviour designer, sound expert,...). The application domain expert, the person experienced in the domain for which the application is built, is not involved very much in doing this task while this would be desirable. Developing a Virtual Reality application requires, without doubt, specialized skills and it is therefore difficult to involve the application domain expert in the development process because of the specific technologies used. An ideal scenario would of course be that a domain expert could model a complete Virtual Environment. Although we admit that this might not be feasible, we want to find a way to at least involve the domain expert more into the development process, and especially in the early phases of the process, the design phase. We propose to accomplish this by means of a novel **conceptual behaviour modelling approach**.

The second objective is closely related to the first. In order to build the high-level specifications describing the behaviour in a Virtual Environment, a set of appropriate **modelling primitives (concepts)** is needed. These modelling concepts will allow specifying behaviour at a high level, independent of any implementation details. In addition, rules on the well-formedness

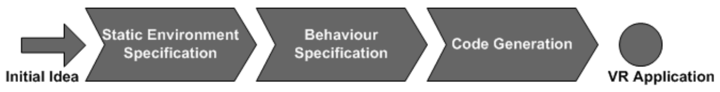


Figure 1.3: *Desired Virtual Environment design process*

of the high-level behaviour specifications are needed which enforce the correct usage of the modelling concepts that were defined. Later on, code can be generated from these high-level behaviour specifications.

Building behaviour specifications using our approach, and using any approach for that matter, shows that very often (and especially when dealing with complex behaviours) parts of a behaviour are re-occurring and in between applications, complete behaviours are re-appearing over and over again. It is therefore useful to provide those common behaviours as predefined components. This prevents people re-inventing the solution over and over again. In addition, optimized implementations may already be available, which you may want to reuse. Therefore, a third objective is to develop a framework that enables the designer to construct behavioural patterns that are involving different objects and behaviours.

In order to validate the overall approach as well as the modelling concepts that were developed, **proof-of-concept software** has been implemented. It will allow to (1) specify behaviour of Virtual Environments, (2) attach it to a particular scene and (3) generate source code from it. Furthermore, this tool can then be used to perform a number of **experiments** in order to assess if people having no skills in VR can model behaviours in a natural way.

1.4 Research Approach

The research approach taken in this dissertation is on one hand to tackle the formulated problems and on the other hand to achieve the aims and goals that were stated above. This is done by extending an already existing conceptual modelling approach with an approach which allows performing the behaviour modelling task. Using this behaviour modelling approach, the designer can describe, at a high-level, the behaviour of a Virtual Environment. It consequently improves the overall development process of Virtual Environments in general and of behaviour in particular.

The specification of behaviours in this behaviour modelling approach is done in two consecutive steps. In the first step, the *behaviour definition*, the behaviour is specified separately from the static scene. In the second step, the *behaviour invocation*, the behaviour is then eventually attached to the scene.

The high-level modelling concepts developed for building such behaviour specifications were searched for using a top-down approach as opposed to a bottom-up approach. This means that we did not start from the existing (low-level) building blocks available in the current Virtual Reality technology (like sensors, interpolators,...), but we have investigated which high-level building blocks are needed in order to describe the possible behaviour that is required. A number of research domains have therefore been investigated and useful modelling concepts have been extracted to finally come to a conclusive set of modelling concepts that can be used to describe the behaviour without falling back to the low-level details of the implementation.

In order to cope with the problem of re-occurring parts in a behaviour and re-appearing behaviours in between applications, a pattern-based approach has been investigated. The behaviour modelling language that is developed is therefore extended with the concept of generative design patterns. This mechanism allows the designer to easily manage the behaviours at a larger scope. A number of behavioural design patterns are made available in the framework and a mechanism is provided to customize them towards the current context.

Two prototype software tools have been developed to validate the proposed approach. The first one allows building the behaviour specifications while the second one is implemented to view the actual outcome from the specifications after generation of the source code. The software is evaluated by means of a case study consisting of a virtual city simulation for which a number of behaviours are modelled. Two other smaller examples will also be presented. A number of controlled user experiments have been conducted. The results of these case studies and experiments allow us to identify issues for improvement that can be taken into account in future versions of the software.

Finally, we will also show that both during the initial design as well as in a later phase of the development process, the information within the high-level behaviour specifications can be used as a means of guiding the designer in the creational process of developing the behaviour. We also show that even the usability of the Virtual Environment itself can be enhanced with this kind of information.

1.5 Significance

As described earlier, the development of a Virtual Environment often lacks a proper design phase, especially for behaviour. Since the behaviour is a key element for having realistic and convincing Virtual Environments, it is important that this aspect is given the necessary attention. Even though the modelling of the behaviour has received more and more attention lately (see the following chapter), there is still room for improvement. We propose an approach to facilitate the design of behaviour for Virtual Environments together with supporting tools. The major advantages, which make this work significant, are the following:

- The behaviour specification is so called action-oriented which means that it focuses on the actions that an object needs to perform as opposed to traditional approaches focusing on the states that an object can be in. Specifying the behaviour in such a way is more natural for application domain experts.

- The specification of the behaviour is completely separated from the design of the static scene and implementation of the Virtual Environment. This provides a clean separation of concerns, which is generally accepted to improve the quality of designs as the designer can consider each design aspect separately.

- The behaviour can be specified independent from the objects that may execute the behaviour and also independent from the interactions that may be used to invoke the behaviour. Hence, it improves reusability since the behaviours can be reused for multiple objects and that the same behaviour can be triggered in different ways depending on the application.

- The developed modelling concepts are application domain independent which makes them suitable to describe any kind of behaviour from any kind of domain. This means that no specific background knowledge in VR is required in order to model object behaviour.

- The use of the behavioural design patterns allows for a better reuse of previously built behaviours as well as of existing algorithms. They also enable faster development and prevent common mistakes. Less experienced designers in VR will benefit from the use of patterns, since they can use them to construct more complex systems even if they do not have the skill to devise the patterns themselves.

Ultimately, the intention of this research is that people having no background in VR will find it easier to specify the behaviour of a Virtual Environment and that they would be able to build dynamic Virtual Reality applications themselves.

1.6 Outline of this Dissertation

This dissertation is organized in 9 chapters. **Chapter 1** has set the general context and stated the problems that we want to tackle. The aims and objectives of this work are explained together with their significance.

Chapter 2 reviews related work within the field of modelling behaviour in Virtual Environments. It starts with an overview of existing high-level modelling approaches for Virtual Environments. It continues with examining approaches that particularly focus on modelling behaviour, classified by their nature in either model-based, scripting languages or dedicated software tools.

Chapter 3 covers the methodology and context used for this research. A brief background on conceptual modelling and a motivation for the use of conceptual modelling into the design process of Virtual Environments is provided. Then, a description of the VR-WISE approach, being the framework used for the research, is given. First, a general overview is given after which a detailed description is provided on each of the different aspects of this approach. Also a short example is given where each aspect is illustrated.

Chapter 4 describes in an informal way, how we have extended the VR-WISE approach to include the modelling of behaviour in a Virtual Environment at a conceptual level. A motivation for the approach taken is given and an overview of the different models introduced (i.e. Behaviour Definition model, Behaviour Invocation model) is given. This chapter is an introductory chapter for chapters 5 and 6.

Chapter 5 builds on top of chapter 4 and provides details on the different concepts introduced to specify both simple and more complex behaviours in Virtual Environments. The chapter is divided into two main parts. In the first part, all the concepts necessary in the Behaviour Definition are discussed. The second part focuses on the Behaviour Invocation and describes all the concepts needed for this model. For each of the concepts, their semantics and their notation are given along with an illustrating example.

Chapter 6 builds on chapter 5 and describes how behaviours can be combined in a structured way in so-called behavioural design patterns. It gives details on the way a pattern is specified in our approach and how they can be used and adapted to fit ones needs. This chapter also describes a collection of patterns that are already supported by our approach and how custom patterns can be built.

Chapter 7 discusses the prototype tool that has been implemented to support the approach. First, a detailed overview of the software architecture of the tool, called OntoWorld, supporting the overall VR-WISE approach, is given. Next, the different parts that have been added or modified in this architecture in order to integrate successfully the behaviour modelling approach are discussed.

Chapter 8 presents the work that has been done in order to evaluate the research presented in this dissertation. To illustrate the approach and to test it, a number of case studies are presented. This will also show the limitations of our approach. Also the setup and the results of usability tests performed are discussed.

Chapter 9 reflects on the research results obtained in this dissertation. A summary is provided; the contributions of this work as well as its limitations are given and discussed in the context of the initially stated research aims and objectives. Finally, possible extensions and further work are discussed.

CHAPTER 2

Related Work

In the previous chapter, the context for the work to be performed in this dissertation has been set. The problems related to the design of a Virtual Environment were listed. The objectives and significance of this work were also introduced. The purpose of this chapter is to present a review of prior work that has been done by others in the field of designing Virtual Reality in general and in particular, with the modelling of behaviour in a Virtual Environment.

The approaches that are described in this chapter are the ones that are well described in literature, in other words, approaches that are stable and described by multiple publications. The general approaches are discussed because they provide a good overview of the different elements that a Virtual Environment modelling approach should contain and more important how the behaviour modelling phase is integrated within the overall approach. The different specific behaviour modelling approaches are discussed because they give us insights in the modelling concepts necessary for modelling behaviour in a Virtual Environment. Here, we investigated true VR modelling approaches as well as animation approaches since this is the direction we are aiming for with our behaviour modelling approach.

The chapter is structured in the following way. Section 2.1 gives an overview of existing design methods for Virtual Environments, which also address the problem of designing VEs from a higher-level point of view. Section 2.2 will discuss approaches that specifically focus on the modelling of behaviour in Virtual Environments. Some additional related works are briefly considered in section 2.3. Finally, in section 2.4, the work presented in this dissertation is being situated in the context of the research field.

2.1 High-Level Design Methods for Virtual Environments

For many years, the development of Virtual Environments has been done in an ad hoc manner without an explicit design phase preceding the implementation. As a result, the systems were often (re-)implemented completely from scratch. Only recently, the problem of a lacking design phase has been acknowledged and this has resulted in a number of high-level design methods. This section will give an overview of these design methods.

2.1.1 VRID

The **V**irtual **R**eality **I**nterface **D**esign (VRID) [Tanriverdi and Jacob, 2001] model and its accompanying methodology provide a guidance to designers in the early stages of the design of Virtual Environments. The research resulted from earlier work conducted towards the specification and programming of the interaction in non-WIMP (Window, Icon, Mouse, Pointer) user interfaces.

The VRID model depicted in figure 2.1a shows the component architecture of a VR interface as identified by the authors of VRID. This model gives the components that address the characteristics of a VR interface together with their inter-relationships. These components are graphics, behaviour, interaction, mediator and communication. The **graphics** component deals with the visual representations of objects. This component is needed in order to render the objects nicely in the interface. Since this approach focuses on the interaction and behaviour of objects, the visual representation, often created by the graphics designer, is being treated as a black box. The **behaviour** component describes the different kinds of behaviours that exist. A distinction is made between *physical behaviours* and *magical behaviours*. The physical behaviours refer to changes that are similar as the ones occuring in the real world (e.g. falling, bouncing of a ball). The magical behaviours refer to changes that have no physical counterpart in the real world (e.g. changing colour when selected). Furthermore, simple (physical or magical) behaviours can be composed to form composite behaviours. The **interaction** component is closely related to the behaviour component and describes how the inputs from external data sources are interpreted and routed to the correct behaviours. The **mediator** component takes care of all the internal communication between the different components in the model. In this way, all the components of the model can be easily replaced without interfering much with the other components. Finally, the **communication** component describes the incoming data from different external sources and the outgoing data from the objects together with the message passing mechanisms used in order to do so.

The model provides the designer with a structured overview of the differ-

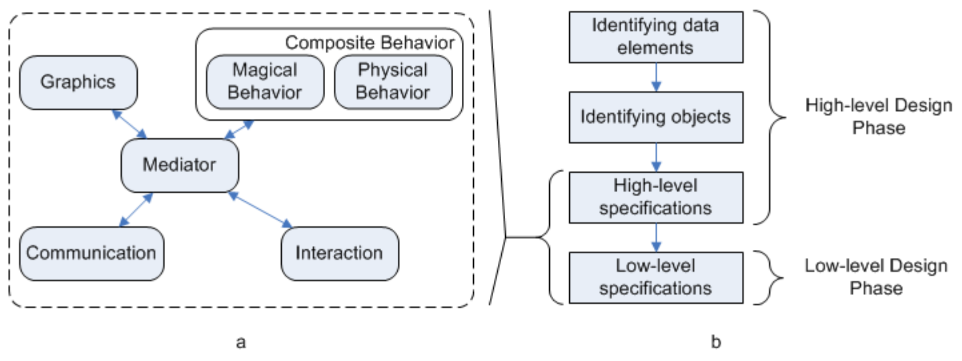


Figure 2.1: *The VRID model (a) and the VRID methodology (b)*

ent aspects of a VR interface. These aspects need to be taken into account when designing VR interfaces. Applying the structure of the VRID model systematically will lead to interfaces being properly designed. An accompanying VRID methodology is proposed (see figure 2.1b) which guides the designer in applying the VRID model. The starting point of the VRID methodology is a document that describes the functionality of the system in natural language. This specification is used as the input of the design process that is divided into two phases:

- The **high-level phase** aims at specifying the interface from a high-level and independent from the implementation. Here, a number of steps are followed. Firstly, the elements exchanging data with the system are identified. These elements can be the user, special devices or other VR systems. Then, the different objects that play a major role in the interface are identified. This can be done using traditional object-oriented analysis techniques and design guidelines. Next, the objects themselves are modelled using five steps. (1) The graphical representations that are required for the objects are described. (2) The behaviours performed by the objects are identified and classified into simple and composite ones. The simple behaviours are further divided into their categories (physical or magical). The way in which simple behaviours are combined into composite behaviours is also expressed here. (3) The interactions are specified between the data elements and the interface and between the objects within the interface. (4) The internal communications are specified in order to avoid conflicts between the messages sent from one component to another through the mediator. (5) The external communications are declared to control the incoming and outgoing data of the interface.

- For the **low-level phase**, the same five steps from the high-level phase described above are repeated, but this time in a more detailed way.

> (1) The graphical models of the objects are linked to the behaviours (physical and magical), that they need to be able to execute. (2) The behaviour is modelled using the PMIW user interface specification language [Jacob et al., 1999]. Data flow models are used to represent continuous behaviours while state transition models represent the discrete behaviours. (3) The interactions are also modelled through the PMIW specification language. (4) Different scheduling mechanisms are selected to resolve any internal communication conflicts that might arise. (5) The different kinds of message passing mechanisms for the communication between the system and external data sources are selected.

The aim of the VRID model and methodology is to decompose the design problem into smaller and simpler problems. In such a way, it becomes easier to develop the different components. This enhances reusability as well as communication between the designers and developers. The methodology also uses a top-down approach going from an abstract high-level representation to a lower-level detailed representation, which is then used as the input for the software developers.

In the high-level phase, the interaction and behaviour can be specified rather intuitively. However, a large part of the high-level phase consists of specifying the communication between the different components which involves more difficult actions to be taken. The low-level phase forces the designer to deal with low-level issues which can be difficult and too complex for non-experienced designers.

2.1.2 VEDS

The **V**irtual **E**nvironment **D**evelopment **S**tructure described in [Wilson et al., 2002] is a user-centred, approach for specifying, developing and evaluating VE applications. The main aim of this approach is to guide the designer in its design decisions in such a way that usability, likeability and acceptability is improved. This will eventually lead to a more widespread use of Virtual Environments.

The approach is the most elaborated high-level design approach found in literature. Figure 2.2 summarizes the overall structure.

The development process starts with setting the initial requirements of the application domain for which the Virtual Environment is to be built (e.g., what functionalities need to be available in this type of application for this particular domain). Here, a clear understanding of VR/VE attributes, what a VE can and cannot do, is needed in order to already make decisions about alternative technologies to use in the system. This will lead to defining the goals of the application, which will be the driving force behind the overall design process. At this stage, the priorities and constraints on the goals are

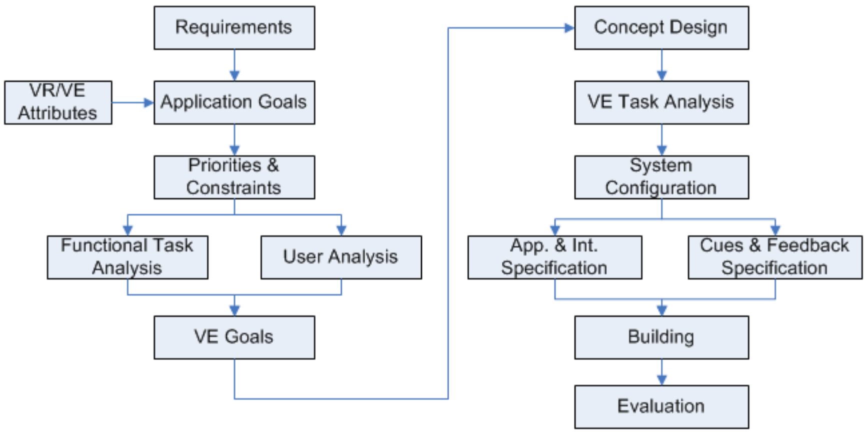


Figure 2.2: *The VEDS methodology*

identified since probably not all of them will eventually be obtained.

The emphasis of the VEDS approach is to have as much direct input as possible from the end-users in the beginning of the design process. Therefore, the initial stages involve assessing which tasks and functions must be completed in the VE, determining the user characteristics and needs (user analysis), and to produce a task analysis. This is done by means of interviews, discussion groups and real world task analysis. The task analysis is often split up into two separate phases, depending on the intentions of the application. In a first phase, the analysis is done for the actual tasks that are modelled (application tasks). In a second phase, which is performed a bit later, the analysis is done for the tasks to be carried out within the Virtual Environment (VE tasks). Based on the analysis phase, a number of specific Virtual Environment goals will have to be provided by the users of the system which will then form the input for the Virtual Environment specification in the next phase.

In the conceptual design phase, the actual Virtual Environment is specified at a high level making balanced decisions on the goals that were set up at the beginning of the process. This specification is used by the developers to build the Virtual Environment. So, it is very important that the specification at this stage meets the requirements of the users. Therefore, it is imperative that all the stakeholders are involved in this phase. If not, there is the risk that the end result will have a high chance of not being acceptable and that it needs to be re-implemented. In VEDS, the storyboarding technique is used as a method to describe, design, outline and agree on the form that the Virtual Environment should take at the end. Storyboards use a terminology that is commonly known by all the stakeholders resulting in a better communication between the different stakeholders and ultimately, leads to a better consistent specification of the Virtual Environment system.

After the specification has been completed, the Virtual Environment can be constructed. This involves making a number of decisions in order to come to a system that best meets the requirements of the user. A first fundamental issue is to decide where to concentrate the modelling effort. Here, trade-offs will have to be made between providing the user with functionality or producing a visually more appealing environment itself. For usability, the most important aspect is the interactivity where a middle ground needs to be found between the accuracy of the modelling and a reasonable approximation of the reality in order to have the best solution. The total number of objects together with their complexity also needs to be carefully considered when including them in the Virtual Environment. One might choose to use techniques to manage the level of detail of the objects or even omit some objects if this increases the overall speed of the application. Besides the interactivity and appearance, also the cues and feedback to the user need to be addressed since they will greatly influence the usability of the system as well. The cues tell the user what interactions are possible at a given moment. The feedback is the response of the system on any interaction that was performed by the user.

Finally, the last phase involves performing an extensive evaluation of both the Virtual Environment itself as well as its usefulness. The evaluations are divided into examinations of validity, outcomes, user experience and process. This means that before the development goes too far, a more detailed examination can be made on how participants will respond to different elements in the VE and utilize its functionality, and be able to understand and work with all the interface elements.

With VEDS, the domain for which the VR application is developed is really integrated into the design stage. Furthermore, there is also a sort of iterative loop in which the design of the VR application is step by step refined until it meets the customer's expectations. Nevertheless, the domain expert is not very much involved into the actual design of the VR application and is solicited only at the beginning of the design phase.

2.1.3 CLEVR

The **C**oncurrent and **LE**vel by Level Development of **VR** systems (CLEVR) approach looks at the design problem from a software engineering point of view and applies current techniques from this field to Virtual Reality design. The CLEVR is the successor of the ADASAL/PROTO approach [Kim et al., 1998].

The authors in [Kang et al., 1998] see a Virtual Environment as a combination of three inter-related aspects: **form**, **function** and **behaviour**. Form is the visual appearance of the objects, together with their structure and the overall structure of the scene in the Virtual Environment. Function is the specification of what objects do to accomplish the behaviour, while behaviour itself specifies how the objects change and carry out different functions. Changing one of the aspects of the Virtual World often affects the other aspects. Therefore, CLEVR urges for a design approach with a concurrent (as referred to in its name) design of the three main components as opposed to the traditional more sequential processes where first the form is being created and afterwards, the function and behaviour are programmed. This is considered as one of the main reasons why designing Virtual Worlds is such a complex activity. CLEVR introduces a methodology to assist the designer in dealing with all three aspects concurrently.

Level-by-Level refers to the fact that the approach is based on an incremental and hierarchical modelling paradigm. This means that the application will be developed, validated and delivered at different stages and will be continuously refined in a top-down approach starting from simple models with few details towards more complex and detailed models. The overall modelling process therefore follows a spiral software development model.

The CLEVR modelling approach is so-called performance driven. Usually, in the first phase of designing a Virtual World, the main focus is on the visual and behavioural aspects. At this point, no attention is paid to the performance of the system yet. It is only in a later phase that more aspects related to VR (such as performance) get some thoughts. After these issues have been treated, the third phase will often address issues related to improving the presence in the Virtual World. Where traditional design methods stop at the first phase, when the resulting Virtual World has been generated, the CLEVR approach goes further and provides additional guidelines to help the designer in the second and third phase as well.

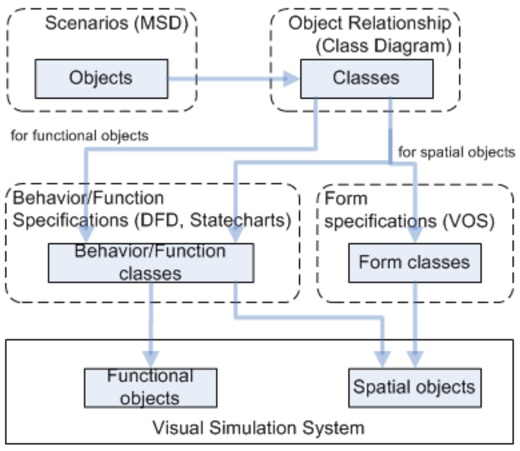


Figure 2.3: *The CLEVR design approach*

The first phase in the approach is obviously the **specification** and is depicted in figure 2.3. As shown, the process starts with a number of *Message Sequence Diagrams* (MSD) which describe the functionalities of the system in terms of signals that are exchanged between the different objects that comprise the Virtual World. This will help to identify the objects that are needed and will be the basis for a *Class Diagram*. The class diagram shows the different objects together with their relationships. The notation that is used is similar to the notation of UML. Next, the behavioural and functional aspect is defined by means of state charts and data flow diagrams respectively. The specification of the form aspect is done in an inter-leaved fashion with the behavioural and functional aspect. It is expressed using the *Visual Object Specification* (VOS) language. The language is similar to C++. A specification for the form of an object resembles to a C++ class definition in which the different properties as well as the spatial constraints are encoded. The following step is to convert the state charts and data flow diagrams into an object-oriented representation so that they can be easily merged with the object-oriented form specification. The overall goal of the first phase is to come to an object-oriented model of the entire system, integrating all three aspects into one whole.

The second phase in the process is the **performance estimation** and **model selection** phase. During the first phase, a number of iterations have resulted in a number of models at different levels of detail. In this phase, a simulation is done to estimate the performance of the application using several combinations of these models. The toolbox in CLEVR includes a specification simulator that reads the models and executes them with some performance data as outcome. Depending on the initial requirements, and based on the performance data, a particular combination of models is selected and the other models will be neglected.

The third phase of the process is the **presence** and **special effects** phase. Here, it is assumed that everything is already designed and tested, and that the only thing left to worry about is some additional features to enhance the overall presence of the Virtual World. At this stage, the designer might add changes to some of the visual representations of the objects either to have more detail or to change some of the behaviours in order to be more realistic.

Although the approach provides a way to design VE applications, it is based on the assumption that the designer understands the UML notation and has some knowledge about Object-Oriented (OO) design. It is very much based on classical software engineering principles. For these reasons, the approach is not very accessible to non-experienced users or the application domain expert.

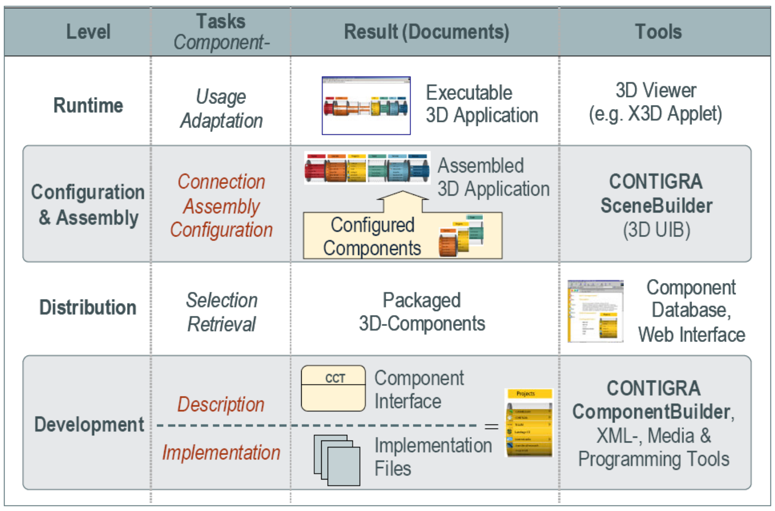
A more detailed description of the approach together with examples can be found in [Seo and Kim, 2002].

2.1.4 CONTIGRA

The **C**omponent **O**rie**N**ted **T**hree-dimensional **I**nteractive **GR**aphical **A**pplications (CONTIGRA) approach is a high-level declarative approach that enables the design of interactive 3D component-based applications, mainly targeted towards the Web [Dachselt, 2001].

CONTIGRA is a so-called document-centred approach which uses structured documents in order to describe the components, what the interfaces are to these components and how they should be combined and assembled into complete 3D applications. For this, a set of different markup languages based on XML, are introduced. This allows specifying the structured documents in a consistent and declarative way. The approach thereby promotes the involvement of non-programmers into the design process of 3D applications. This is in contrast with the code-centred approaches to create component-based 3D applications where the necessary programming skills are required.

Figure 2.4 shows the different levels, together with the corresponding documents and the tools that are involved in building 3D applications using the CONTIGRA approach. A short overview of the development stages will be given here. For a more detailed description, the reader is referred to [Dachselt et al., 2002].



Level	Tasks Component-	Result (Documents)	Tools
Runtime	*Usage Adaptation*	Executable 3D Application	3D Viewer (e.g. X3D Applet)
Configuration & Assembly	*Connection Assembly Configuration*	Assembled 3D Application; Configured Components	CONTIGRA SceneBuilder (3D UIB)
Distribution	*Selection Retrieval*	Packaged 3D-Components	Component Database, Web Interface
Development	*Description*	CCT Component Interface	CONTIGRA ComponentBuilder, XML-, Media & Programming Tools
	Implementation	Implementation Files	

Figure 2.4: *The CONTIGRA levels and tasks*

The construction of the components is the first step that needs to be performed at the **development** level. This level is divided into two phases, the implementation phase and the specification phase which respectively use the **SceneGraph** language and the **SceneComponent** language.

The SceneGraph language is an extension of the X3D language and its purpose is to describe the component at the lowest level, in terms of its geometry and behaviour. The geometry is being represented by an ordinary **X3D** scene graph while the behaviour is being represented by a **Behavior3D** graph [Dachselt and Rukzio, 2003]. The Behavior3D concept has been introduced to cope with the shortcomings and unnecessary complexity of X3D's built-in support for behaviour. Behavior3D uses an object-oriented paradigm. New behaviour nodes can inherit from other behaviour nodes and possibly extend them. Alternatively, new behaviour nodes can also be constructed by referencing and using other behaviour nodes. The built-in interpolator nodes from X3D are wrapped into Behavior3D nodes so they too can be used in the behaviour graph as any other node. In addition, two special nodes are included, namely *Sequential* and *Parallel*, which allow for an intuitive combining mechanism of nodes in order to construct complex Behavior3D nodes. A Sequential node activates all containing nodes in a sequence while all the child nodes of the Parallel node are activated at the same time. Besides, the standard (wrapped) nodes coming from X3D and the two special nodes, also external scripts and Java code extractions can be easily integrated using the same language.

The SceneComponent language is the main language of the architecture. This allows defining the interfaces of the components separated from the actual scene graphs. Not only the accessible parameters and methods are declared here but also the component can already be configured as a particular instance of a component. This follows a sort of prototype-based paradigm where a document at this phase is being seen as a prototype. Later on, it can be copied and changed to suit the needs of the application at hand. Furthermore, the language also allows specifying complex components by combining a number of smaller sub-components.

Both the SceneGraph document (together with all the external files such as scripts, audio files, picture files,...) as well as the SceneComponent document form one single CONTIGRA component. The people that are involved in the development level are usually experts in the field of 3D and/or Virtual Reality, or people that at least have some programming skills. This level is the most time consuming but once a large number of components have been specified using the two dedicated languages, it will drastically reduce the time needed to build complete 3D applications.

The second level is concerned with the **distribution** of components on the Web. The corresponding task is that the developers need to search and retrieve the different components that are required for a particular application. To facilitate this, a kind of Web portal needs to be built that allows querying an online collection of components in a flexible way.

Once all the separated components are gathered, they need to be integrated into one whole which is done at the **configuration** and **assembly** stage. The dedicated **Scene** language is introduced with the purpose of configuring and assembling the different components. A document in this language represents a specification of one single interactive 3D/VR application. At this point, the specification is still in a technology-independent format.

The final step is then to transform this independent specification into a specific 3D technology so that it can be viewed at the **runtime** level in a 3D browser or executed as a standalone application.

As opposed to other approaches, in the CONTIGRA approach it are the developers who are really involved in the initial stages of the design process. They are actually building the different components. The end-user is only involved in a much later stage where the components just need to be linked to each other and instantiated.

2.1.5 SENDA

According to the **SENDA** approach [Sanchez-Segura et al., 2005], Virtual Environments can be seen as a special kind of information system (from the software engineering point of view) and thus have to be designed as such. Hence, this involves the standard analysis, design and implementation phases encountered in software engineering.

The SENDA framework defines a more formal process model for developing Virtual Environments. The idea is to refine traditional software processes in order to improve the quality of Virtual Environment development. A number of ISO standards (12207 and 1074) that are specifically targeted for software engineering have been modified to better suit the development of Virtual Environments. It is a combination of reusing existing techniques and introducing new techniques that are required to deal with the context of Virtual Environments.

In general, the overall SENDA process model bears much similarities with traditional process models used in Software Engineering (see figure 2.5). The framework comprises 11 different processes and each of these processes is subdivided into tasks to come to a total of 36 tasks which are needed to develop a complete Virtual Environment. Not all of these tasks will be discussed in detail here, for this the reader is referred to the corresponding literature.

The framework starts with the *management process* which consists of a *planning* process and an *estimation* process like it is the case in the more traditional approaches. Here, an initial planning is made of what is going to be done when as well as an estimation of the time, resources and costs of the development for this particular Virtual Environment.

After this process has been completed, the actual *development process* commences which is again divided into three different sub-processes.

- Firstly, an *analysis* process is executed in which the requirements for the Virtual Environment are considered. Here, a **questionnaire** must be filled in order to identify the tasks that need to be performed in the

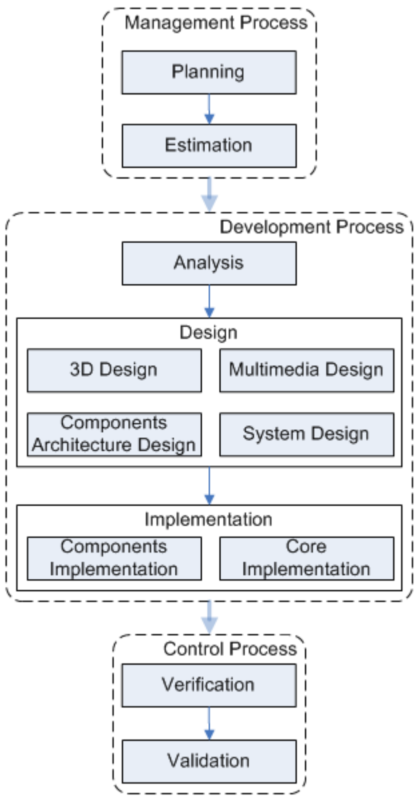


Figure 2.5: *The SENDA process model*

end result. These tasks are specified more formally through **use-cases** (known from UML) describing the user interaction with the system. In addition, **use-concepts** is a mechanism they have introduced to describe the system functionalities that are not triggered by means of user interaction. Afterwards, an initial static and dynamic modelling task is performed.

- Secondly, the *design process* is done. This is divided into four sub-processes. The *3D design* process is meant to describe how the environment is going to look like. This is achieved by means of two specification, one specifying the VE itself, and one specifying the objects inside the VE. Furthermore, there is also a map visualizing the spatial location of the objects and a navigation diagram representing the links between sub-spaces in the VE. The *multimedia design* process involves determining the different multimedia elements such as sounds, images, animations and so on that will be needed in the VE. The *components internal architecture* process is meant to define the actions (behaviours and interactions) that can take place within the VE. An interesting issue about this approach is the use of **interactive behaviour patterns** to facilitate the design of components comprising a Virtual Environment [Sanchez-Segura et al., 2004]. The *system design* process mainly consists of extending the static and dynamic models developed in the analysis process. This involves **class diagrams** for the static modelling and **transition diagrams** for the dynamic modelling. In this step, also a prototype of the user interface of the actual application is developed.

- Thirdly, in the *implementation process*, the actual Virtual Environment is implemented in two steps, the *components implementation* process and the *core implementation* process. In the former, all the different components are implemented independently from each other. In the latter, an empty Virtual Environment is created and it is gradually built-up by adding the smaller components from the previous process.

The framework finishes with the *control process*. The main purpose of this last set of processes is to evaluate the finished product before it is actually delivered to the customer. It consists of a *verification* process and a *validation* process, where really the designs as well as the implementations of the different aspects of the Virtual Environment are submitted to be reviewed.

The SENDA approach is a very formal approach based on existing design approaches used in the software engineering domain. The approach involves a wide range of people into the process such as software designers and graphics designers as well as psychologists, and so forth, all with their own expertise. However, the downside of this approach is that the actual end-user of the application is only involved at the beginning of the process and then only very little.

2.1.6 Ossa

The **Ossa** system is an approach to conceptually model Virtual Reality systems [Southey and Linders, 2001]. Ossa (which stands for "skeleton" in Latin) provides a modelling environment that allows building strong underlying conceptual models, as a sort of skeleton for the Virtual Reality application. These models are a combination of conceptual graphs and production systems. The conceptual graphs are used for representing the knowledge of a world that is about to be designed. The production systems approach is used to capture the dynamics of the application. A brief overview of the architecture is given here, a more detailed description can be found in [Southey, 1998].

The architecture of the Ossa approach is depicted in figure 2.6. The figure shows the two different components of a production system. These

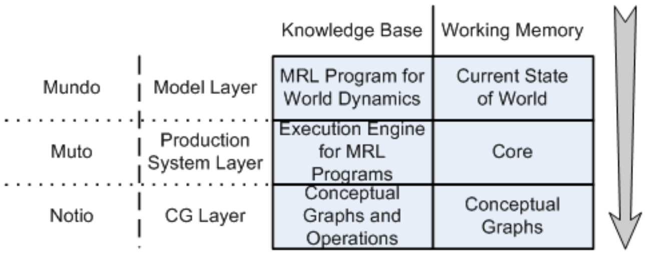


Figure 2.6: *The Ossa design architecture*

components are used as the major modelling technique in the approach, given by the columns in the figure. These are the **Knowlegde Base** containing all the rules and the **Working Memory** containing all the facts.

The Ossa system is further divided, both conceptually as well as in the implementation of the overall architecture, into three different layers as given by the horizontal rows in the figure:

- The **Mundo** layer is used to describe the Virtual World at a conceptual level. This is done by means of specifying the different concepts together with the relationships between those concepts via the conceptual graphs. The behaviour is then described by writing a set of rules in a dedicated rule language called the **Muto Rule Language** (MRL). When designing a new Virtual World, the designers only need to create the conceptual model in this layer and do not need to know anything about the internal workings of the underlying layers. However, this layer is depending on the second layer for its execution.
- The **Muto** layer is responsible for managing the conceptual graphs that make up the current state of the Virtual World (core). Furthermore, it also deals with evaluating and executing the different rules, described using MRL, inside the knowledge base which will then modify the facts in the core. As a result, the Virtual World is moving from one state into the next one. This layer depends on the Notio layer for its functionality in order to accomplish this task.
- The **Notio** layer provides the basic data structures for the facts that are encoded via conceptual graphs. It allows representing the conceptual graphs together with the functionalities for storing, retrieving and manipulating them. Since the rules are also described by means of conceptual graphs, it also provides the ability to perform some operations on conceptual graphs as required by these rules. It is built on top of the Notio API, a general purpose API written in Java that is used for developing conceptual graphs tools.

As it can be seen, conceptual graphs form the basis for the entire design approach that enables an intuitive specification of the Virtual World. The Ossa approach uses conceptual graphs, both as a representation of the conceptual model as well as an internal data representation mechanism in the system itself. The production systems approach for the dynamics allows for the behaviour to be specified in considerably less lines of code than other object-oriented approaches and it facilitates future maintenance. Furthermore, since there also exists a widely accepted graphical notation that can capture the conceptual graphs and the production rules, the designer can visually specify the conceptual model, which greatly enhances usability of this approach.

The disadvantage of the Ossa system is the large complexity it brings since it is not using a normal procedural approach for specifying the dynamics. A rule-based approach is used resulting in more complicated execution patterns. Besides this, the rules need to be described in a kind of logic programming style which is not so accessible to non-programmers.

2.1.7 I4D

The lack of a proper design methodology is also acknowledged in the research performed in the context of the **i**nteractive **4D** (i4D) framework [Geiger et al., 2001]. I4D is a framework for the structured design of all kinds of interactive and animated media. The approach not only targets the domain of Virtual (and Augmented) Reality but also the domain of 3D graphics and multimedia.

The i4D design approach aims at expressing the conceptual models in terms of concepts that are familiar to all the stakeholders of the application. Therefore, an actor-based metaphor is chosen where the different components of the Virtual Reality application are seen as actors in a play. The actors act like particular roles that are specified by the designer. The approach extends current metaphors from the television and film industry to capture the dynamical aspects of a Virtual Reality application that were previously not supported. This metaphor is a sort of abstraction over the basic scene graph but which is a suitable representation to enable communication between the different designers in the process. At the same time, it contains enough details to make sure that a future mapping to an implementation is possible.

In order to bridge the large gap between what the user requests and the actual implementation, a step-by-step approach needs to be followed. An overview of the design process is given in figure 2.7. The different steps of this approach are:

- **Scenario descriptions** provide a number of short stories behind the Virtual Reality application as if it was described by the stakeholders, knowing nothing about technical issues involved in VE design. It gives

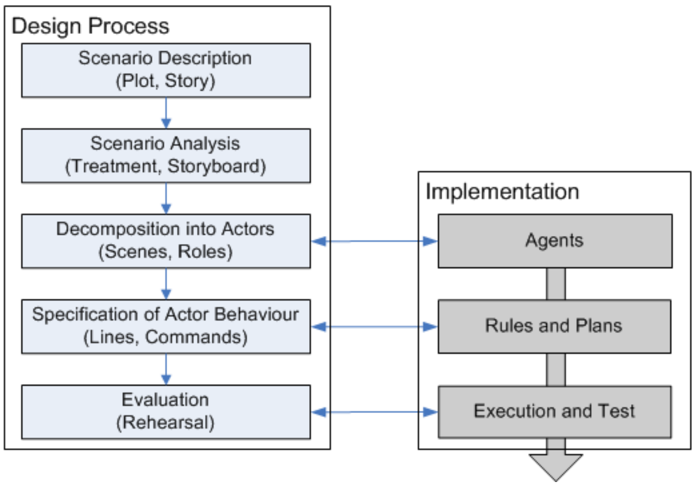

Design Process
Scenario Description (Plot, Story)
Scenario Analysis (Treatment, Storyboard)
Decomposition into Actors (Scenes, Roles)
Specification of Actor Behaviour (Lines, Commands)
Evaluation (Rehearsal)
Implementation
Agents
Rules and Plans
Execution and Test

Figure 2.7: *The i4D design approach*

details on which kind of objects there are, how they look like, how the users interact with the objects and what the results will be afterwards. This specification is specified in natural language.

- In the **scenario analysis**, the different scenarios are further refined into structured situations where more details are added about the interactions and the animations of the objects. This results in a more formal representation of the scenarios (e.g., a storyboard).
- A **decomposition** of the storyboard is done into well-defined scenes. The different actors are identified and (possibly) hierarchically structured. Finally, a direct mapping is made between the actors and their geometry they control on the one hand and their actions they perform on the other hand.
- The **behaviour specification** formalizes the behaviour of the actors. Behaviour that has been designed earlier will just be linked to an action coming from the animation library. For behaviour that has not yet been encountered, a new action needs to be defined.
- In the **evaluation** step, the application is executed several times to see if the behaviour performed by the actors corresponds to the roles that were specified in the scenarios. If not, the decomposition and behaviour specification step need to be iterated.

Next to the design approach, the i4D framework also provides a number of tools covering the whole design process from idea to the actual implementation of the Virtual Environment. An important aspect is the **high-level component-based library** for 3D content, the core of the i4D framework.

This is a library that also follows the actor-based metaphor and allows the easy design of 3D actors and actions through a high-level programming API. On top of this library, a scripting language is provided that enables interactive design and a number of visual tools are developed to enhance the design even further. The framework is extendable by other (external) components that serve different purposes such as simulations, physics, etc. It allows managing and using different components more easily, which is a strong requirement in most Virtual Reality systems nowadays. A more detailed overview of the framework is given in [Geiger et al., 2000].

The actor-based metaphor which is used in the specifications in i4D only allows describing a Virtual Reality application using the domain terminology from this particular domain, namely that of role plays. Other domain knowledge cannot be used and therefore, it is too restrictive in the sense of being able to handle many different domains. Furthermore, most of the things eventually need to be programmed in their framework, which is a kind of abstraction of the currently existing graphics libraries.

2.2 Behaviour Modelling Approaches

Whereas the approaches introduced in the previous section deal with the design of Virtual Environments in general, this section will in particular discuss research that is related to the aspect of modelling behaviour for Virtual Environments. The section is divided into three sub-sections. First, the approaches based on a graphical notation are discussed, then the approaches using textual scripting languages are considered and finally, some dedicated software applications are discussed. The focus is on the approaches most closely related to the work presented in this dissertation.

2.2.1 Graphical Notations

2.2.1.1 Petri Net Script

The **Petri Net Script** (PNS) described in [Blackwell et al., 2001] is a graphical language addressing the specification of behaviour of virtual actors. It provides a graphical interface to behavioural scripting. PNS is built on Petri Nets, a formal language that was originally created for the specification of concurrent, discrete-event dynamic systems. A Petri Net usually consists of *places* representing the states and *transitions* representing possible changes between states. The places and transitions are connected by a number of arcs. At any moment in time, the network can be given a state by marking its places with tokens. Since Petri Nets do not allow storing data within the tokens and do not support hierarchical structuring of the networks, they are not really suitable for specifying complex behaviour. In order to overcome these problems Petri Nets have been extended into so-called High-Level Petri Nets [Jensen and Rozenberg, 1991]. However, their main drawback is that they are less easy to use and thus increase the learning time. The PNS expands on the principles used in existing High-Level Petri Nets and in order to enhance ease-of-use and flexibility. Firstly, PNS introduces the concept of *external places* (e.g., sensors). The places may also contain embedded information in addition to only keeping record of the tokens. PNS also extends the usual token typing system by arranging token types in an intuitive object-oriented hierarchy. Furthermore, the arcs can be labelled with a number of expressions representing *pre- or post-conditions*. Finally, PNS allows structuring the networks hierarchically to allow making abstractions and reduce the complexity. As a result, Petri Net Script makes complex behaviour specification more accessible and allows exploring the full capabilities of a real scripting language in a more intuitive way. However, the Petri Nets were initially not designed for the modelling of behaviour typically encountered in Virtual Environments. Therefore, concepts that were initially not designed for modelling behaviour in a VE need to be used for exactly this purpose which is not a natural way of modelling.

2.2.1.2 Flownets

In [Smith and Duke, 1999][Willans, 2001][Smith et al., 2000], Virtual Environments are considered to be hybrid systems and the behaviour is modelled as a combination of discrete and continuous components. The **Flownet** formalism introduced by the authors is a specification formalism designed for Virtual Environments. In Flownets, the discrete processes of the behaviour are described using traditional Petri Nets. However, as discussed in the previous approach, some modifications have also been made here, i.e. the tokens are able to represent conditions and a distinction is made between normal transitions and *interaction transitions* in order to cope with priority handling in case of concurrent firings. Furthermore, the concept of *inhibitor arc* is developed to specify that a place does not need to contain a token for a transition to be fired. Since the Petri Nets lack capabilities for describing continuous processes, a number of modelling constructs have been added coming from System Dynamics research. The *continuous flow* concept represents a flow of information that is considered as a continuous stream. A *plug* concept allows sending and receiving a continuous flow of information. The *flow control* element can allow the flow of information to be passed or not based on the incoming arcs. A *store* concept is developed to act as a sort of equivalent for the place concept but then for continuous information. Finally, a *transformer* has been added which can modify the information on one or more continuous flows into one or more resulting flows. The Marigold toolset, which has been developed to support the approach, provides a means to specify the Flownets rather independent from the current context (e.g., input/output devices, world objects). A stub of code is generated which can then be seamlessly integrated with the rest of the Virtual Environment. However, the graphical notation does not allow to hierarchically structure the models, to decompose them in sub-models. The consequence is that the models become large and difficult to be read. Therefore, this is not suitable for inexperienced users.

2.2.1.3 Behavior Transition Networks

Another framework that aims at simplifying the design of behaviour is described in [Houlette et al., 2001][Fu et al., 2003] where **Behavior Transition Networks** (BTNs) are used, which are actually generalizations of finite state machines (FSMs), to accomplish this. The BTNs consist of states, representing the actions, and transitions, controlling the flow from one action to another based on a condition, just like in the FSMs. However, a BTN can also contain *variables* that keep information about the current state and can communicate to other BTNs through a *blackboard*. Furthermore, two major extensions have been made to the basic FSM model. First, the representation supports a hierarchical behaviour model. An author can decompose more complex behaviours into a few high-level behaviours, each capturing a portion of the original behaviour. This results in a set of nested behaviours, which are easier to understand and to modify. Behaviours are able to invoke other behaviours just like any predefined primitive action. Secondly, each behaviour may have a number of specializations representing different kinds of ways for performing a behaviour with a similar goal. Polymorphic extensions can be made for every variation of a behaviour, which are all indexed by means of a set of *entity descriptors*. The entities also have these descriptors and the selection of a particular behaviour happens by finding the closest match between both descriptor sets. This is related to the strategy design pattern in software engineering where a different strategy is chosen based on the context at a particular moment. If no match is found, the default behaviours will be selected. The state machine approach is very useful for a particular kind of behaviours, mostly discrete behaviours, but they are less useful for continuous behaviours. Furthermore, it is difficult to describe behaviour of objects that is somehow related to other behaviours of other objects.

2.2.1.4 VisualVRML

The work presented in [Arjomandy and Smedley, 2004] acknowledges the difficulties in specifying behaviour in VRML (X3D) and introduces a system where behaviours of objects can be visually specified. In this system, the designer is able to drag-and-drop icons from a palette onto the workspace. The icons, representing VRML nodes that are being supported, are classified into four categories: Objects, Sensors, Interpolators and Scripts. Each node is displayed by means of a rectangle containing an icon that identifies the category and additionally an icon denoting the type of the node within this category. Connectors are attached to the nodes on the left and right side corresponding to the input and output fields respectively. The input and output connectors can be linked to each other using arrows as the equivalent of the ROUTE statements. The system presented here is merely a one-to-one correspondence with the behaviour facilities available in the VRML language. Every element in VRML has a graphical counterpart in this system. Hence, it does not really provide an additional layer of abstraction. Considerable knowledge about the VRML language is required to build behaviour specifications and a lot of scripting is still involved.

2.2.1.5 Rube

The **rube** modelling framework [Hopkins and Fishwick, 2003][Fishwick, 2000][Kim and Fishwick, 2002] goes one step beyond the visual (2D) programming paradigm and provides developers with a way to specify dynamic models in three dimensions (3D). Where other methods use fixed 2D, or textual representations for their modelling concepts, rube allows the designer to create models in 3D without being constrained to a predefined set of (graphical) representations for the modelling constructs. In other words, the behaviour of the objects is defined using another set of objects. The modelling process of rube is as follows. The designer starts with the initial scene containing the objects for which the behaviour needs to be designed. Then a particular dynamic behavioural model is selected and the dynamics and interactions between different models are specified in case there are multiple models selected that need to exchange information. A number of dynamic behavioural model types are supported by rube including Finite State Machine, Flow Charts, Petri Nets and many others for which special templates have been created in VRML. Next, an appropriate metaphor is selected. For this, rube allows designers to create their own 3D representations, which are based on the metaphors. When choosing the objects, it is important to pick the ones that have a meaningful relationship with the actual modelling concepts. The visualization of the metaphors gives a kind of semantic clue of its actual functionality. Afterwards, the mappings are created between the metaphorical objects and the actual modelling concepts from the dynamic behavioural model. In the last step, the models together with the mappings are combined into the final model. The rube framework provides a means to intuitively specify the behaviour in a Virtual Environment based on personalized 3D metaphors. The main target audience for rube is the intermediate to expert developer since some general knowledge about the behavioural model types is required together with a good knowledge of VRML.

2.2.2 Scripting Languages

2.2.2.1 Alice

The **Alice**[1] [Conway, 1997][Conway et al., 2000] project presents an easy-to-learn scripting language, together with an authoring system. It provides, to a broader audience of end-users, a means to describe behaviour of 3D objects without any skills in 3D graphics or animation techniques. On one hand, the scripting API presented in Alice draws a lot from usability engineering. Data is gathered from usability tests with the system that uncovers issues that are taken into account the next time the API is refined. On the other hand, it is being inspired by spatial understanding literature. Literature that describes the abstractions people use to describe 3D scenes, and the techniques people use in 3D wayfinding tasks. Navigation has also been extensively used in the development of the API. As a result, the API in Alice is found to be more intuitive than some existing 3D APIs. In order to program behaviour, Alice provides a rich set of primitive commands identified by names which are carefully chosen having strong relationships with natural language (e.g., move, turn, pan,. . .). Furthermore, the traditional names for the coordinate axes in the commands, namely X, Y and Z, are removed from the API, and replaced with the more useful direction names and surface names of forward/back, left/right, up/down. Alice is also said to be time-based which means that animations are specified to take a particular number of seconds as opposed to the frame-based system where they are to take a particular number of frames. Using overloaded methods, supporting several different calling patterns for a single command, provides a controlled exposure of power to the user. Novice users can employ the simple default commands while more experienced users can use the more advanced features of the system. Primitive commands can be combined to form more complex animations through the use of two commands: DoTogether and DoInOrder for respectively simultaneous and sequential execution. The most interesting aspect about the Alice system is that it provides novice users with a tool to start programming behaviours in 3D. In a sense, our system has many similarities with Alice. However, as acknowledged by the creators of Alice it is still too difficult to model complex animations and in some cases, too much syntax is needed to specify things.

2.2.2.2 Improv

Improv [Perlin and Goldberg, 1996] is a system designed for the scripting of interactive actors in Virtual Environments. It follows the expert systems philosophy and provides the authors with a means to control the choices that the actors make and how the actors move their bodies accordingly.

[1] http://www.alice.org

The system firstly consists of the animation engine that uses specifications of the animated actions to manipulate the bodies of the actors. The actions are specified by means of setting a set of value-pairs for each of the degrees of freedom in the model of the actor. These values are ranges of poses corresponding with changes over time. The exact degrees of freedom values are then at runtime computed by means of time-varying noise signals that can be seen as a sort of interpolators. The designer can place the created actions into groups based on the fact whether they can be executed simultaneously or not. The order of the groups determines the execution priority of the actions. The second component is the behaviour engine that is responsible for the more high-level behaviours and decisions about which actions to invoke. Since the behaviours in Improv are mainly driven by interactions with the user, which cannot be predicted in advance, fixed sequential animations are not sufficient anymore. Instead, layers of choices from the more long-term plans to the more short-term activities must be created. These layers must take into account the non-deterministic response of the user as well as the always-changing state of the environment. The designer can write scripts that allow an actor to randomly choose between a set of behaviours. Optional weights associated with the behaviours will affect the probability that a particular behaviour is chosen. Additionally, designers can create more complex decision rules in which case the choice to favour particular behaviours over others is based on their outcome. Each decision rule consists of a list of factors all associated with a weight, which control how much influence this factor has on the decision. The factors can be information from either the actor itself, from other actors or from the environment in general. For all this, Improv uses an English-like scripting language to allow non-programmers to create powerful interactive applications. It is true to say that the Improv system is quite powerful at the higher levels. However, in the lower levels, the basic primitive actions are still described using the most fundamental modelling constructs. This, together with the need of the noise signals, makes this system rather unintuitive for non-specialists. Furthermore, it only focuses on modelling human-like figures.

2.2.2.3 STEP

A bit related to the previous works is the language called **S**cripting **T**echnology for **E**mbodied **P**ersona (STEP) [Huang et al., 2003a]. STEP is a simplified and user-friendly, but still expressive, scripting language for 3D web agents based on H-anim[2], a standard for representing humanoids in VRML/X3D. The main focus of the language is on communicative acts, and in particular the ones that involve geometrical changes of the body parts. The reference systems used within STEP are based on those from the H-anim specification. The standard pose of the humanoid was used as an inspiration to define

[2]http://www.hanim.org

a direction reference system on top of the traditional coordinate reference system. For the three dimensions, more intuitive natural language-like directions are introduced. Besides the time reference system of VRML/X3D, a more flexible system was defined by introducing the notions of beat and tempo. The body reference system is the same as the one in H-anim and the different names of the joints and segments can be used to refer to the parts of the body within the actions. Usually, a complete behaviour in STEP is described as a posture together with a movement. Turning body parts implies setting the rotations for the relevant joints. Moving the body means setting the position of the whole node hierarchy. Turn and move are the two main primitives for body(-part) movements. A number of typical operators, to either compose actions into more complex ones and to interact with the environment, are included. In the first group, there are `seq` and `par` for respectively sequential and parallel execution of the actions. In addition, there is `choice`, which is used for a non-deterministic execution and `repeat`, which denotes a repeated action. In the second group, there are `if_then_else` and `until` for conditional executions of the actions together with constructs to get and set the state of the world. STEP is a high-level declarative scripting language based on Dynamic Logic. In [Huang et al., 2003b] the XSTEP language, an XML-based markup language, is proposed as the successor of STEP. In the same way as Alice, the STEP approach also provides a very intuitive means to specify behaviour. Since it is based on the H-anim standard, it can, however, only be used to describe the behaviour of virtual characters. It does not provide enough modelling concepts to be usable for modelling all kinds of behaviours and is not restricted to a particular sub-domain.

2.2.2.4 Smart objects

The work presented in [Kallmann, 2001][Kallmann and Thalmann, 2002] introduces a feature-based modelling approach to define so-called **smart objects**. This means that all the possible information that is related to the behaviour of the objects is defined within the object itself rather than programmed separately. A smart object can be compared with an object, an instance of a class in object-oriented programming, holding data belonging to the object together with providing methods for manipulating this data. The different features of a smart object are described in a text-based script file. The features that are supported are divided into four categories. Firstly, the intrinsic object properties, properties that are a part of the object design such as the movement specification of its moving parts as well as any physical properties. These specificationo are created using actions which are the simple functions that an object(-part) can perform, like translations, rotations, etc. Secondly, the interaction information is additional information that helps the actor in the interaction with the object. This is done by either specifying positions or gestures. Thirdly, the object behaviour, that describes how the object reacts to the interaction with the actor, is specified. And fourthly, the expected actor behaviour describes the behaviour that the actor has to perform in order to enable the interaction with the object. Both the object behaviour as well as the actor behaviour is specified through predefined behaviour plans consisting of a set of primitive operations. The operations can invoke the actions, change the states of the object and call other (sub-)behaviours. The language used for the behaviour definitions has many similarities with finite state automata. The writing of this kind of scripts works well for simplistic behaviours, but it does no longer work for more complex behaviours. Therefore, the concept of behaviour templates has been brought to life. The templates contain the most commonly used behaviour definitions, which can be plugged into other behaviour definitions to form more complex ones. Still, much programming is needed which is not suited for people having no skills in this area. Furthermore, since everything is included inside the object itself, there is little separation of the different aspects. In general, this is constraining the reusability.

2.2.2.5 BDL

In [Burrows, 2004][Burrows and England, 2005], a declarative **B**ehavior **D**escription **L**anguage (BDL) is developed that is based on VRML and supports both object-object behaviour and user-object behaviour. A BDL specification starts with declaring the behaveable objects either by specifying them in a VRML-like syntax or by loading existing VRML objects from external files. Additionally, some type information can be given. Then, behaviour can be attached to the object. This can be done within the object specification or external to it. Such a behaviour specification basically gives a list of the set of stimuli together with the required responses referring to them through their names. The stimulus is an externally created object that is capable of responding to a number of events coming from user actions, other objects, or the world itself. In the simple case, there would be just one response of a single stimuli but the BDL language allows more flexibility in this regard. That is, one stimulus can trigger multiple responses; multiple stimuli can trigger a single response. The response is a code module, which is also defined outside the specification file. Responses can on their turn also trigger other stimuli or can send messages to another object in the scene. In addition, the behaviour specification can contain conditions that must be met in order for the response to be executed if the stimuli are fired. It is important to note that objects do not have behaviours coded into them directly. The specification and implementation is completely separated. The scene specification file just links objects to the required response, which makes that the specifications are easy to understand and to be modified later on. Again, the focus went to attaching behaviours to the objects and specifying the events that trigger them, resulting in a very intuitive alternative way in comparison of the VRML way of handling things. However, the behaviours, i.e. the actions that are executed by the objects, are still described using the standard Java programming language.

2.2.2.6 PAR

The approach presented in [Badler et al., 2001][Badler et al., 2000][Kipper and Palmer, 2000] introduces **P**arameterized **A**ction **R**epresentation (PAR) as a conceptual specification mechanism for the actions of virtual agents. PAR is designed to bridge the gap between the behaviour specification, specified through natural language instructions, and the virtual agents performing the behaviour. A PAR gives a complete specification of a particular action. It is called parameterized because the action depends, for details, on how it should be executed, on the agents and also the objects involved. The basic form of a PAR resembles the standard SVO (Subject-Verb-Object) sentence in natural language. The subject and object relate to the agents and objects respectively in the PAR. However, certain sentences also incorporate additional information, to enrich the action that is described, in the form of adjuncts added to the sentence or within other sentences in proximity of the main one. PAR includes slots for many types of information of this kind. There are some applicability and termination conditions that specify what needs to be satisfied in order for the action to be respectively executed and terminated. Then, there are also the preparatory specifications that describe the actions which need to be executed before the current action can be executed. Finally, there are also the execution steps that allow specifying what really happens in the action. In case of a complex action, this can contain a list of sub-actions that are either executed in sequence, in parallel or in any combination. Usually, a PAR can be specified at two different levels analogous to class and object in object-oriented programming. There is the non-instantiated PAR. This PAR contains all the default information of a PAR except the ones of the agents and physical objects involved in the action. In other words, the parameters are not yet filled in at this level. Then, there is also the instantiated PAR that contains the pointers to the agents and physical objects involved. If some new information is specified here, it will override the default one specified in the non-instantiated PAR. Natural language input for describing the behaviour is difficult since firstly, many of the natural language parsers are still not yet perfect and thus introduce a lot of errors. Secondly, most people use different sentences for describing the same things. As far as we could see from the consulted literature, the system is not able to cope with this.

2.2.2.7 HPTS

In [Devillers et al., 2002] and [Donikian, 2001], an approach is described to design adaptive and flexible behaviour for any entity involved in a Virtual Environment and interacting with other entities. The proposed language is completely based on the **HPTS** (Hierarchical Parallel Transition Systems) formalism and data-flows. It allows the specification of state machines that can be structured hierarchically and which can be activated simultaneously together with the communication that might occur between those state machines. A specification of a state machine is basically done as follows. First of all, the machine can be parameterized which allows setting the initial characteristics of the state machine. Next, variables can be declared with a distinction between local variables and public variables, i.e. available to parent state machines. The body of a machine consists of a list of states together with a list of transitions between those states. A state is defined by its name and its activity in terms of data-flow executed once this state is activated. A transition is defined by an origin, a transition expression and the write-expression. The transition expression consists of a read-expression containing the conditions that have to be met in order for the transition to be fired. The write-expressions are the resulting events that will be generated and the activities that are executed on the state machine. The transition will eventually lead to the execution of the next state(s) in the state machine. In [Devillers and Donikian, 2003], the work has been extended towards a more high-level specification of complete scenarios. A scenario is decomposed into a number of smaller sub-scenarios and will at the lowest level consist of a sequence of elementary tasks. An elementary task is being described by including pure C++ code into the specification. However, some instructions have been added to the language allowing the specification of the tasks to be done more easily. The scheduling of this hierarchy is done through the usage of temporal relationships. In contrast with the BTN approach, the HPTS approach really uses the state machines at the programming level. Nonetheless, this is resulting in similar problems as the ones encountered in BTNs, namely not being able to specify continuous behaviours well and lacking capabilities to synchronize behaviours. Furthermore, there is the problem of the lack of programming expertise with novice users. Therefore, it is very difficult to be used by novice users for modelling behaviours other than the most trivial ones.

2.2.2.8 ScriptEase

Another interesting piece of work is described in [McNaughton et al., 2004b] and [McNaughton et al., 2003] where an approach is presented to allow behaviours to be created using template-based Artificial Intelligence (AI) behavioural patterns. **ScriptEase** is developed in the context of game design. In ScriptEase, the process of defining a pattern goes as follows. Firstly, a list of situations is constructed. These situations are composed by means of so called Atoms, which are little pieces of scripting code. There are five kinds of atoms supported in the approach. Event atoms describe which events might occur in the game world the script is responding to. Action atoms are wrapping the code that is executed in order to change the game world after an event has been thrown. Besides the list of parameters, an action atom also contains a function name together with the code body that actually implements what the action is meant to do. Definition atoms wrap the code for gathering information from the game world (e.g., defining local variables). Condition atoms use information from the game world and prevent some actions to be executed based on a predefined condition. Option atoms simply encapsulate the enumeration types of the scripting language denoting different fixed options that can be set by the designer. Once the pattern has been created, it can be used in many different instances. An instance of a pattern is made by attaching the pattern to (a) particular actor(s). Finally, the patterns need to be adapted according to the current context. The software tool that supports the ScriptEase model is completely menu-driven and all the options for the behaviours and scenarios are given in natural language, which greatly reduces the need for programming skills. This work is focussed on AI patterns used in game programming. Nonetheless, because of the close relationship between current game technology and Virtual Reality, this approach could be directly applied to the design of Virtual Environments in general and parts of the work presented in this dissertation have been inspired by this approach. There is still a great deal of programming knowledge required to have a full understanding of the system although the patterns are fully parameterizable via a GUI interface. The introduction of a graphical notation for the patterns would seriously enhance the approach.

2.2.3 Software Applications

2.2.3.1 Virtools

The **Virtools**[3] software suite [McCarthy and Callele, 2001][Nahon, 2005] is a development environment designed to produce interactive 3D applications and deploy them on a variety of platforms. Virtools is not really a modelling application; however, some basic functionality is available through its authoring tool. In general, the 3D content must be created elsewhere and imported to create the static scene. The main focus of Virtools is on the design of object behaviour, which is facilitated by means of a graphical language. A behaviour consists of a header and a body. The header contains the name of the behaviour and the owner being the behavioural object to

[3]http://www.virtools.com

which this behaviour is attached. The body is described through a schema. The basic modelling concept used to design behaviours is the Behaviour Building Block (BB). BBs can be seen as a visual representation of a function as known in programming languages. Each BB can have zero or more Behaviour Inputs and Behaviour Outputs. A BB is executed after it has received an activation from the input and it will send an activation through the output once the execution is done. The BBs are connected through Behaviour Links describing the order in which they are executed. In addition, a BB can have a number of parameters. There is a distinction between input and output parameters, which receive and transmit data. Parameter Links are used to direct the output of one BB to the input of another BB. A number of BBs can be composed to form a Behaviour Graph (BG). These graphs are treated exactly the same as a BB but they can be saved and reused as a black box later on. Furthermore, the concept of Message was introduced to enable BBs to change the state of the environment or to notify other behaviours. Virtools comes with a standard library of predefined behaviours. When a desired functionality is not available in the library, it can be created using the built-in scripting language, or by creating a new building block using Virtools' SDK.

The downside of using Virtools is the complexity of the software tool itself. Furthermore, the function-based mechanism (where the designer needs to take into account the frame-to-frame basis way of processing the behaviours by the behaviour engine) tends to be less comprehensible for non-VR-experts.

2.2.3.2 3D Studio Max / Blender

Two frequently used authoring tools are 3D Studio Max [Murdock, 2002] and Blender [Roosendaal and Selleri, 2005]. Since both are very similar in nature, they will be discussed together. Firstly, different types of animations are natively supported by these tools. A first type of animation is through the use of key frames. Here, a series of complete positions and orientations are saved, one for each of the key moments in a set of units of time (frames). An animation is created by interpolating an object fluidly through the frames. A second type of animation consists of using motion curves. Here, curves can be drawn for each XYZ component for location, rotation, and size. These form the graphs for the movement, with time set out horizontally and the value set out vertically. The third type of animation is to use path-like animation where a curve is drawn in 3D space, and the object is constrained to follow it according to a given time function. Besides the normal animation, most of the tools also support the creation of so-called armatures. They were initially developed for animating characters but they can also be used to animate regular objects (and mechanical structures). To build an armature object, the designer will interconnect a number of "bones" which make up a sort of skeleton. This skeleton is linked to the real objects in the Virtual Environment. In order to manipulate the armature (and the objects related to it), one can then create different poses that describe the objects' changes and the constraints by means of specifying actions. In order to cope with some missing functionalities or more complex behaviour, a third option is provided by most software packages. This is the use of a dedicated scripting language such as MAXScript or PythonScript for respectively 3DS Max and Blender.

An advantage of the authoring tools is that they, in a sense, automate the programming and therefore free the designers from being dependent on programmers. However, this only works well for simple behaviours. In order to model complex behaviour, usually some kind of scripting language must be used, requiring programming skills.

2.3 Other Related Work

Besides the different high-level design methods that have been discussed above, there are a number of other design methods available. In [Fencott, 1999], a design methodology is proposed based on the work presented in [Kaur, 1998]. Another one is for example the **JADE** approach which makes use of adaptive components to build dynamic Virtual Environment systems [Oliveira et al., 2003]. In [Molina et al., 2005], the **TRES-D** methodology is proposed combining design issues from some of the approaches discussed here.

Other works concerning the modelling of the behaviour, and not discussed here, can be found as well. For example, there is the **beh-VR** approach [Walczak, 2006] using the concept of VR-Bean as a reusable component controlled by scenario scripts. The approach described in [Messing and Hellmich, 2006] uses aspect-oriented programming techniques to define the behaviour and weave it together with the scene. A closely related set of markup languages are **AML** and **CML**, initially designed to control virtual characters [Arafa et al., 2002].

Very little research is available on applying design patterns to model the behaviour in VEs. Most related work concerning design patterns is found in the field of game design. However, current games share many characteristics with VEs and hence those concepts from game design could prove to be very useful in the design of VEs [Clarke-Wilson, 1998].

The work described in [Diaz and Fernandez, 2000] acknowledges the difficulties encountered in VE design. It proposes an object-oriented model to design the different aspects of a VE consisting of rooms, avatars and objects populating the VE which are all considered as objects interacting with each other. Based on this model, a number of design patterns are presented which support the designer into solving recurrent design problems in the context of designing the virtual space, the mobility and behavioural issues. The patterns called Area and Gate deal with the structural design of VEs. The Locomotion and Transport pattern deal with navigation. Finally, the Collector is used to model elements behaving in the same way depending on another element. This work discusses implementation issues as well which can be readily used by programmers in their design. The downside is then that a great deal of programming knowledge is required which makes its usefulness towards non-programmers very small. Our approach tries to build an additional layer of abstraction which allows the designer to visually construct the patterns without any programming knowledge.

The *Game Design Patterns project* [Bjork and Holopainen, 2004] studies computer games in terms of interaction, components and design goals with the intention of creating the basis for a common language for game designers. The overall goal of the project is to investigate how the design of computer games can be facilitated, similar to our goal of facilitating VE design. In order to do this, over 200 design patterns are developed covering most aspects of gameplay. The most interesting ones, for our purpose, are those dealing with the temporal aspect. It covers patterns about different actions and events that might occur in a game and that change the internal state of the game, as well as those dealing with the interaction between the player and the game environment and between the objects in the game environment. However, for the moment, they only use the patterns as a means of communicating the design and thus do not discuss any implementation issues. This poses constraints on the active use of these patterns for the development phase of the games. Since the patterns are focused on game design, they describe problems which are oriented towards very concrete situations in games. Most of these situations rarely occur in VEs in general. However, many of them could be abstracted from in order for them to be applicable in VEs.

In [Folmer, 2006], a number of design patterns are presented with the focus on making games more accessible on one hand and lower their development costs on the other hand. The patterns are divided into two categories. Firstly, the usability patterns introduce concepts that help improving the usability of the game and also the interaction of the player with the game. Secondly, the accessibility patterns try to overcome the difficulties that people with disabilities encounter when playing the games. Although the focus is less on the design, some of the patterns could be useful for the design of VEs.

Another interesting piece of work is the one described in [Cavazza et al., 2004a] and [Cavazza et al., 2004b]. Modelling every physical event in a VE would be very computationally demanding. In order to keep a high level of realism, the actual behaviours of the objects are separated from the physical processes involved in the interaction with these objects. The events related to interaction are hence descretized through the definition of so-called programmed system events. Afterwards, they can be effeciently captured by means of a descrete event-based system typically found in game engines. The behaviour itself of the objects in Virtual Environments is specified by means of Qualitative Physics. Basically, the system consists of describing the objects together with all its properties and then relating these objects to qualitative processes in which they might be involved. In essence, such a physical process is described by means of qualitative variables (representing the states of the process) and the relationships between those variables by means of influence equations and qualitative proportionalities. The approach allows for a high-level specification of physical behaviour over the analytical computations currently used in physics engines.

A detailed literature review of behavioural animation and behaviour modelling is given in [Cerezo et al., 1999] and [Millar et al., 1999] respectively. These works discuss a wide variety of techniques that are also used in the field of modelling the behaviour in Virtual Environments but which were found less relevant for the work described in this dissertation. A fairly recent state-of-the-art report concerning toolkits and software applications used for modelling Virtual Reality applications can be found in [Pellens et al., 2006a].

2.4 Situating this Dissertation in the Field of Modelling Behaviour

Before positioning the work presented in this dissertation within the context of modelling behaviour for VEs, it is interesting to notice that the current high-level design methods have a strong emphasis on the modelling of the behaviour and less attention is paid to the design of the objects itself. This proves the statement that was made earlier that the specification of the behaviour in Virtual Environments is a very challenging task and many people are still looking on how to make improvements.

Looking back to the general high-level design methods introduced earlier in this chapter, it is also interesting to notice that in most of the cases there is a very loose coupling between the behaviour specification and the rest of the aspects of the Virtual Environment. In some cases, this is made possible by using separate diagrams for describing the behaviour. In other cases, it is enabled through the use of an independent behaviour graph notation, in analogy with the scene graph, next to the usual scene graph for the visual aspect. These separated specifications are, in a later phase, translated (or merged) to come to the final result. This way of working reduces the risk that a change in the behaviour specification will have a large impact on the specification of the other aspects. Furthermore, the specification of the behaviour is strongly based on the other aspects of modelling the Virtual Environment since every aspect needs to be modelled by taking all the other aspects into account in order to come to the best possible solution. The proposed behaviour modelling approach will need to consider these issues as well when it is integrated with the overall design approach described in chapter 3.

Both the model-based as well as the text-based approaches have their advantages and disadvantages. The model-based approaches allow the specification of behaviours in a more intuitive way, without the necessity of having programming skills. Nonetheless, the models are often expressed in a formalism coming from a particular domain (e.g., Software Engineering,...), which requires the designer to have some background knowledge about the modelling concepts used in these domains. Furthermore, complex behaviour often results in very complex models, which quickly become difficult to read and to be understandable. The difficulties of the scripting-based approaches are that they require some programming skills or at least some background knowledge about scripting languages. Therefore, they exclude non-programmers from designing behaviour. A textual interface involves a lot of typing which will drastically slow down the design process. However, they provide more flexibility to the programmer and enable the specification of more complex behaviours. This then leads to larger code listings making the maintenance and the reusability more problematic.

In this dissertation, a novel approach for designing the behaviour is presented. The approach can be primarily categorized as being model-based since high-level conceptual models, represented by means of graphical diagrams, are used to describe the object behaviour. Although the approach started off to be completely model-based, it evolved into a combination of being partly model-based and partly text-based. This resulted in a mixed graphical/textual notation as it will be explained in more details in the main chapters of this dissertation (chapters 4, 5 and 6). It is our aim to come to a behaviour specification approach that can deal with the same complexity than the text-based approaches while retaining the intuitiveness of the model-based approaches.

The work presented here can best be situated in the area of animation/scripting rather than in the area of physically based modelling. The behaviours that can be defined with our approach are predefined animations. This dissertation does **not** deal with modelling the physics. It is not our intention to compete with highly accurate physical models (e.g., natural phenomena,...). As described later, in chapter 7, a physics engine is included in the proof-of-concept software. Its functionality is used as a black box. The physics itself is not modelled explicitly.

Furthermore, in this work, it is assumed that all the objects (or parts of objects) are solid objects, meaning that the distance between any two points of an object remains constant, as opposed to flexible objects which are deformable, meaning that the relative positions of points of the objects can change. Therefore, our approach does not enable manipulations at the vertex and edge level but only allows geometric manipulations.

CHAPTER 3

Methodology

The previous chapter discussed the related work. It started with reviewing a number of general high-level design methods for developing Virtual Environments. It continued with focusing on explicit behaviour modelling approaches since this is the main topic of this dissertation. The purpose of this chapter is to give a general introduction to the high-level design method, called VR-WISE, which is used as the context for the work presented in this dissertation.

The remainder of this chapter is structured as follows. The VR-WISE approach is a conceptual modelling approach, therefore, section 3.1 first elaborates on conceptual modelling and gives the motivation for using this approach. Furthermore, VR-WISE is also so-called ontology-driven as it uses the concept of ontology as a representation formalism. The concept of ontology and why this representation formalism was chosen is discussed in section 3.2. Afterwards, in section 3.3, a detailed description of the VR-WISE approach is given. Section 3.4 briefly places the work presented in this dissertation into the overall VR-WISE approach. Section 3.5 provides a short summary of this chapter.

3.1 Conceptual Modelling

As described earlier, most Virtual Environments that are developed today are designed in an ad hoc way. Low-level specification and programming languages are used to respectively model the static and dynamic part of the Virtual Environment. Unfortunately, these types of languages are too low-level to describe Virtual Environments in a way that is readily understandable by a variety of users, and in addition they lack the essential high-level abstraction mechanisms that needed when dealing with the development of large or complex systems. What is needed is a higher-level specification of the Virtual World, also called a *conceptual model*. A conceptual model can roughly be defined as a simplification of (a part of) reality. It models a portion of the 'real world' that is of interest in a particular application domain, also called Universe of Discourse (UoD). It is independent of any implementation, the target technologies, programming languages, or any other physical considerations [Dillon and Tan, 1993]. Conceptual modelling or conceptual design is referred to as the process of constructing such a high-level specification (i.e. conceptual model).

A specification like this is generally expressed by means of a conceptual modelling language containing appropriate high-level modelling concepts. The language, as well as the different modelling concepts, needs to adhere to some criteria:

- **Expressiveness.** It is important that the expressive power of the modelling concepts is sufficient to allow capturing all the features of the domain so that the resulting models can also be used as input for the implementation phase.

- **Clarity.** The meaning of the modelling concepts should be intuitively clear so that they can be easily learnt and remembered. This means that these moodelling concepts should be expressed in a terminology that is familiar to most of its users.

- **Formal Foundation.** The language needs to have a formal ground in order to ensure unambiguity and executability (e.g., to automate the storage, verification, and transformation of the models).

While developing the high-level modelling concepts and the associated graphical notations, presented in this dissertation, these criteria were always kept in mind.

It is widely acknowledged that conceptual modelling is a prerequisite for successfully designing applications. Examples of that can be found in the Software Engineering domain and in the Information Systems domain. In Software Engineering, the design of an application needs to assess as many characteristics as possible in order for it to be of any assistance in the implementation with the hope that the final result is meeting the initial requirements set up by the client. The best known conceptual modelling language is UML (Unified Modelling Language), which enables the system-analyst to make an object-oriented design of a software application in all its facets, without being restricted to a particular programming language [Braude, 2000]. Next to the Software Engineering domain, the domain of Information Systems has also a long history in conceptual modelling. In an information system the emphasis is on the conceptual modelling of the data.

It needs to be modelled carefully in order to ensure that the data needed will be available. This must be modelled in a consistent and in a correct way. In fact, database technology actually only became common practice after the introduction of a conceptual modelling phase into the overall design process. The well-known conceptual modelling languages for information systems are ER (Entity Relationship) Model and ORM (Object Role Model), which allowed the designer to specify the database at a conceptual level; free from any implementation details [Halpin, 2001]. Because VR applications are more complex than databases or classical software applications, it is our belief that the introduction of a conceptual modelling phase for the development of Virtual Environments will facilitate the use of Virtual Reality technology and make it available to a much broader public including people having no expertise in programming or relevant VR technologies.

This brings us to the question whether or not the existing conceptual modelling languages used in the domains of Software Engineering and Information Systems could possibly be used in (or extended for) the Virtual Reality domain as well. Hence, these languages were investigated thoroughly [Pellens et al., 2004]. This investigation showed that both ORM and ER have enough expressiveness to partly model the static part of a Virtual Reality application. However, as they are initially designed for data modelling they have strong limitations in terms of describing the behavioural part and the interaction part of a Virtual Reality application. UML on the other hand is capable to deal with all three aspects, i.e. modelling the scene, modelling the behaviour and modelling the interaction. The static part for example can be modelled using its class diagram. It can also model the behavioural part of a Virtual Reality application by means of the statechart diagram and the interaction can be described by using the sequence diagram. Although UML could be used to model an entire Virtual Reality application from a software engineering point of view, it is not appropriate because it is lacking expressiveness and intuitiveness towards Virtual Reality design. Therefore, none of these conceptual modelling languages are really suitable to design a complete Virtual Reality application in an intuitive way.

A number of candidate conceptual modelling paradigms to describe Virtual Environments have been briefly documented in [Bernier et al., 2004], all with their advantages and disadvantages. In the VR-WISE approach, a relational-graph oriented paradigm is chosen to design Virtual Environments at a conceptual level through the use of ontologies.

3.2 Ontology-Driven Design

Ontologies play an important role in the VR-WISE approach. They are used for two different purposes. (1) Ontologies are used explicitly during the design process for specifying the high-level conceptual models. (2) Ontologies are also used as underlying representation formalism (internally). Therefore the VR-WISE approach is said to be an ontology-driven approach.

3.2.1 What are ontologies?

Ontology (with capital "O") originated from philosophy where it is being referred to as the study of existence. This discipline tries to answer questions of what can be said about the world, what exists and what does not or what features are exactly identifying objects. Later on, the notion has been taken over by the computer science community in order to represent knowledge of the world at hand. Since it would be impossible to represent everything from the real world, this was often narrowed down to a part or aspect of the world (which is often called domain) sufficient enough for the intended purpose of the application or a family of applications (hence the widely used term *domain knowledge*). The main idea in representing this knowledge is creating an abstract set of objects, concepts, and other entities as well as all the relations that may hold between those entities which are assumed to exist in a particular application domain, that is called a *conceptualization*. This leads us to Thomas Gruber's definition of ontology (with lowercase "o"), which is without doubt the most cited one in the literature:

> *An ontology is a formal, explicit specification of a conceptualization.* [Gruber, 1993]

A detailed interpretation of the definition and other definitions is given in [Guarino and Giaretta, 1995]. In short, an ontology comprises the definitions of the concepts, individuals, properties and relations which make up the conceptualization. Furthermore, the conceptualization has to be formal and explicit, that is, it needs to be explicitly defined in a kind of formal language.

Numerous ontology languages have been developed in the past, ranging from RDF(s)[1], OIL, DAML+OIL[2] and finally the most recent one OWL[3] (Ontology Web Language). In our approach, OWL was chosen as our main ontology language since it is the richest one in terms of already available modelling primitives and it is currently the standard W3C ontology language.

3.2.2 Why using Ontologies?

Initially, ontologies were used to improve communication between either human beings or computers. They serve as a perfect middle ground between plain text (which is understandable by human beings) and programming

[1]http://www.w3.org/RDF/
[2]http://www.w3.org/TR/daml+oil-reference
[3]http://www.w3.org/2004/OWL/

code (which is interpretable by computers but not by most humans). However, the use of ontologies to build conceptual specifications brings about many other advantages as pointed out in [Uschold and Jasper, 1999] and [McGuinness, 2003]. The most important ones are:

- The **communication** between people is improved since they reduce the conceptual and terminological confusion by providing a uniform view of the domain under consideration.
- It enhances the **inter-operability** since different users/applications are using the same set of terms to describe their knowledge. An ontology can also assist in the translation between different representations.
- They give us support in **reusability** since they can be used and reused as a kind of component amongst different software applications.
- They facilitate the creative process of building a **specification** since they are expressed using terminology of the domain the designer is most familiar with.
- Because they are formal by nature, it becomes possible to perform automatic **consistency checking** up to some degree resulting in more reliable applications.
- The rich information contained inside the ontologies can be exploited and additional information can be inferred to enable **searching** and **querying** of the ontology as shown in [Kleinermann et al., 2005].

3.3 VR-WISE Approach

The VR-WISE approach was first introduced by De Troyer et al. in 2003 [De Troyer et al., 2003]. The acronym stands for **V**irtual **R**eality - **W**ith **I**ntuitive **S**pecifications **E**nabled. The aim of the approach is to include an explicit conceptual modelling phase into the overall design process of Virtual Environments. It enables the designer to specify a Virtual Environment through intuitive high-level specifications using his own terminology. There is no need to consider any implementation details; hence the approach does not require any Virtual Reality background knowledge whatsoever. These specifications are then used as input for a (semi-)automatic implementation phase. The ultimate goal is to facilitate and shorten the overall development process of Virtual Environments by means of conceptual models.

Although, there are quite a number of ways to develop VR applications (as mentioned in previous chapter), skilled people are often required in the process. The problem in most approaches is that the Virtual Environment one wants to create must be expressed in terms of (combinations of) low-level building blocks of the VR technology (also called target domain). More people could be involved in the development of Virtual Environments if it would not have to be described using such low-level and specialized terminology. To date, none of the currently available VR development tools allow specifying the Virtual Environment in terms of the domain for which the application is developed (also called problem domain). A known fact is that VR experts are usually not experts in the domain for which the Virtual Environment needs to be developed and vice versa, the domain experts and the end-users are not experts in the field of Virtual Reality. The VR-WISE approach, using ontologies as a specification mechanism, allows the domain experts to be involved and exploit their knowledge by allowing them to use domain terminology in order to specify the Virtual Environments. This is making the design more intuitive and natural for them. The VR-WISE approach inherently bridges the gap between the (high-level) problem domain and the (low-level) target domain.

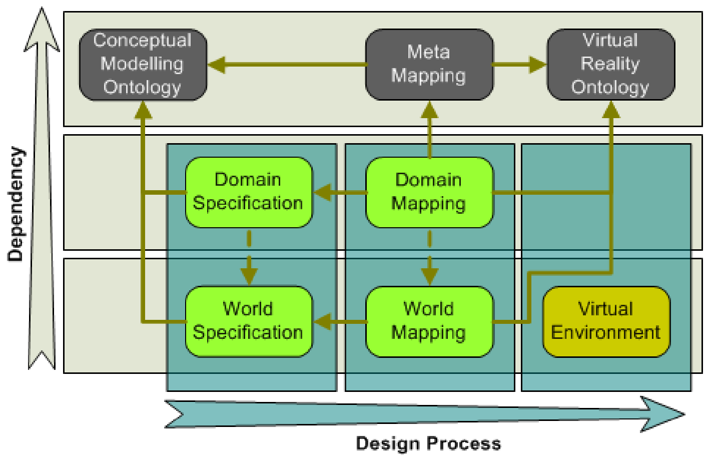


Figure 3.1: *General overview*

Figure 3.1 is a simple schematic overview of VR-WISE. It basically has two dimensions as given by the large arrows and the colours used. Read vertically (in grey), it depicts the different layers in the ontology architecture of VR-WISE, the lower ones depending on (or using concepts from) the ones above. Read horizontally (in blue), the control flow of the design process of VR-WISE is depicted. The remaining part of this section presents the VR-WISE approach based on these two dimensions. An example is used as a way to illustrate the different parts. Section 3.3.1 describes all the ontologies used in the approach starting with the most general ones working our way down to the most specific ones. Afterwards, section 3.3.2, gives an overview of the different phases that need to be performed in order to build a Virtual Environment using VR-WISE. Note that the approach described here differs a bit from the one described in [Bille et al., 2004b],[Bille et al., 2004a] and [Pellens et al., 2005a] since it takes into account the latest improvements made to the approach.

3.3.1 Architecture of Ontologies

Figure 3.2 provides a more detailed overview of the complete ontology architecture. The architecture has three levels namely the ”Meta Level”, the ”Domain Level” and the ”Instance Level”.

- The **Meta Level** has three modules namely the Conceptual Modelling Ontology, the Meta Mapping and the Virtual Reality Ontology. At this level, the ontologies are independent of a particular application domain.

- The **Domain Level** provides the type-level specifications and involves the Domain Specification and the Domain Mapping. The ontologies at this level are dependent of an application domain but are not dependent of a specific application.

- The **Instance Level** defines the object-level specifications and involves, in analogy with the Domain Level, the World Specification and the World Mapping. Here, the ontologies are related to a particular application in a particular domain.

The next section explains the three levels with the associated ontologies into more detail. The ontologies all have been created using the Protégé-OWL[4] editor.

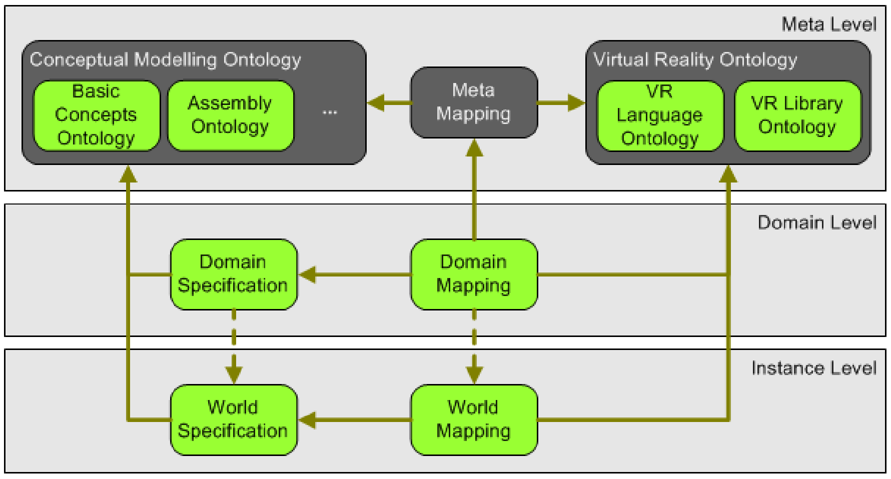


Figure 3.2: *VR-WISE ontology architecture*

[4]http://protege.stanford.edu/

3.3.1.1 Conceptual Modelling Ontology (Meta Level)

In order to model a VR application, a number of high-level modelling concepts are provided. These modelling concepts are independent of any application domain and are defined in a so-called upper ontology [Valente and Breuker, 1996]. This ontology characterizes the meta-level because it acts as a kind of repository of our VR modelling concepts. This ontology is called the *Conceptual Modelling Ontology*. In fact, it is a collection of ontologies. Modelling a Virtual Environment involves many different aspects: the objects in the world, their properties and composition, their behaviour, the interaction between objects and the interaction with the user, etc. For each of these aspects, a different ontology has been created.

Since the Conceptual Modelling Ontology is based on OWL, some basic modelling primitives provided by OWL are already available. The most interesting ones are the `Class` and `Property` constructs that allow defining a type and their properties respectively. Furthermore, OWL has a rich set of characterization primitives for these constructs. It also has the `subClassOf` and `subPropertyOf` relations for specifying generalization hierarchies of respectively classes and properties. The full specification of OWL can be found online[5]. In order to cope with future compatibility issues and to ease the integration of external domain knowledge, it is imperative that these native primitives are used instead of defining our own (see later).

Additionally, the *Basic Concepts Ontology* provides a number of general modelling concepts. The most important one is the class `Concept`; all the domain concepts that will be defined must be defined as a subclass of this class (see Domain Specification). It also holds modelling concepts such as the basic dimensions of concepts (e.g., `Width`, `Height`, `Depth`,...), the appearance of concepts (e.g., `Colour`, `Texture`,...) as well as information about units of measurement. It also defines the different reference frames used within our approach.

The *Assembly Ontology* defines concepts to specify how objects can be related or connected to each other in order to form either unconnected complex objects or connected complex objects.

- The *spatial relations* are used to specify how an object is located in space in relation to some other object. The spatial relations are divided into directional relations and orientation relations [Gapp, 1994][Frank, 1996]. The directional relations are used to specify the position using intuitive directions (e.g., `left-of`, `in-front-of`, `above`,...). The orientation relations are used to specify the orientation of an object by indicating which side of the object is oriented towards which side of another object. Besides the directional and orientation relations, there also exist a great number of topological relationships. A classification

[5]http://www.w3.org/2002/07/owl

has been proposed in [Egenhofer, 1995][Zlatanova, 2000]. However, these relationships are underspecified to be usable in the context of positioning objects in space. Therefore, we chose not to include these relations in our approach.

- The *part-whole relations* allow to specify how an object is composed of different (part) objects. Research in this regards has been done in the field of mereology [Varzi, 1996]. Here, an attempt is made to set out the general principles underlying the relationships between a whole and its constituent parts. This is typically resulting in relations such as `is-part-of` and `has-part`. However, modelling the real world accurately needs more than just the ability to say that one object is part of another object. There are different ways to make a connection and thus several different kinds (subtypes) of the general part-of relation have been identified (e.g., `component-integral`, `place-area`, `member-bunch`,...) [Winston et al., 1987][Odell, 1994]. These concepts were included in our approach. Different taxonomies have also been proposed by [Gerstl and Pribbenow, 1996] and [Iris et al., 1988] but these were not adopted at this moment.

- The Assembly Ontology is currently extended in the context of another PhD research [Bille et al., 2005]. On the one hand there are the *boolean operators* for modelling complex shapes and on the other hand there are the *connection relations*, *constraints*, and so on, for representing mechanical constructions.

The complete specification of the Conceptual Modelling Ontology will not be given here, but it can be found online[6]. One of the aims of this research work was to extend the Conceptual Modelling Ontology with concepts that can be used for specifying object behaviour in a Virtual Environment.

3.3.1.2 Virtual Reality Ontology (Meta Level)

The *Virtual Reality Ontology* is similar to the Conceptual Modelling ontology, a kind of repository of VR concepts but this time the ontology is more presentation- and implementation-oriented instead of modelling-oriented. The ontology focuses on defining the low-level building blocks used to represent Virtual Environments. It describes all the building blocks towards which the conceptual specifications can be mapped (see later). It is also a combination of different ontologies.

The *Virtual Reality Language Ontology* is basically a one-to-one mapping of the X3D node elements to classes in OWL. It defines the class `VRConcept` denoting an object that is directly representable in the Virtual Environment. It also describes the primitive geometries (as subclasses of the `Geometry`

[6]http://wise.vub.ac.be/ontoworld/CMOntology.owl

class) such as `Box`, `Cone`, `Cylinder`, `Sphere`, `Text` together with their main attributes that are necessary to specify them. It describes the appearance such as `Colour` and `Material` as well. Furthermore, it contains concepts like `Transform` and `Group` in order to create complex objects. At this stage, only a limited subset of the X3D language has been defined into OWL, just enough to describe basic scenes. Please note that it is not the intention to develop a complete ontology for Virtual Reality in order to describe a Virtual Environment in all its aspects. This would be a separate subject.

However, incorporating and relying only on the primitive geometries provided by a particular language (in our case X3D) to represent all the objects is too restrictive if one wants to accurately model Virtual Environments of a considerable complexity. What is needed is a collection of more advanced pre-built geometries that can be used like any other primitive geometry. The main purpose of the *Virtual Reality Library Ontology* is to provide such a collection. An object in this collection, a `LibraryObject`, is defined as a special kind of VRConcept (as a subclass thereof). It holds a reference to a file that is containing the advanced geometry (in the case of X3D, it contains a reference to a ProtoDeclare definition) and obviously, also its required attributes are defined. The problem is still that it takes time to create such a library but once it has been done correctly, it seriously reduces the time that is needed to build a Virtual Environment.

The complete specification of the Virtual Reality Ontology can also be found online[7].

3.3.1.3 Meta Mapping (Meta Level)

The *Meta Mapping* is a small ontology consisting of the concepts that are required to enable the mapping (translation) between the high-level specifications using the concepts from the Conceptual Modelling Ontology and the low-level specifications using the concepts from the Virtual Reality Ontology. Mappings are expressed by means of a source and a target. The source must be a subclass of `Concept` and the target must be a subclass of `VRConcept`. This is expressed by the class `ConceptRepresentation` which has a `hasSource` property of which the value needs to be of type `Concept` and a `hasTarget` property that requires a value of type `VRConcept`. Besides the source and target, the class also has some `hasAttributeRepresentation` properties, each containing a reference to an `AttributeRepresentation`. The class `AttributeRepresentation` expresses the mapping of the attributes and is also defined as having a `hasSource` and `hasTarget` property. An optional `hasRule` property takes a value of type `Rule` and defines how the source value is being transformed to the target value. The rule typically consists of a mathematical expression relating the two together (e.g., the

[7]http://wise.vub.ac.be/ontoworld/VROntology.owl

source is three times the target) allowing to describe more complex mappings than just one-to-one mappings.

The Meta Mapping makes sure that the mappings defined later on are structured correctly. The specification of this ontology can also be found on the Web[8].

3.3.1.4 Domain Specification (Domain Level)

The purpose of the *Domain Specification* is to describe, at a conceptual level, the concepts (comparable to object types in Object-Oriented modelling) available in the domain under consideration for the application. The ontology describes the domain concepts by means of their properties as well as their relationships. It is when this ontology is created, that the background knowledge of the domain expert is exploited. During this task, also (information from) existing ontologies can be reused which may drastically speed up the building process of the Domain Specification.

The example below gives an extraction of the Domain Specification that has been developed in the domain of department stores. It defines a concept `&vin;Wine` as a standard Class in OWL. The concept is also defined as a subclass of the `Concept` class (defined in the Conceptual Modelling Ontology). Furthermore, it has a property `hasBottleDiameter` defined. This property is required, as given by the `cardinality` element, and it needs to contain a value of type `Diameter`, as given by the `hasValue` element. Other properties can be defined as well but are not shown here.

```
<owl:Class rdf:about="&vin;Wine">
    <rdfs:subClassOf rdf:resource="&cmo;Concept"/>
    <rdfs:subClassOf>
        <owl:Restriction>
            <owl:onProperty rdf:resource="#hasBottleDiameter"/>
            <owl:hasValue rdf:resource="&cmo;Diameter"/>
            <owl:cardinality rdf:datatype="&xsd;int">1</owl:cardinality>
        </owl:Restriction>
    </rdfs:subClassOf>
    ...
</owl:Class
```

Note the use of the `rdf:about="..."` syntax in the class-definition. In this particular case, it allows us to refer to, and possibly extend, a class defined elsewhere. In a typical case, one would define the concepts from scratch and then the `rdf:ID="..."` syntax would have been used. Here the class refers to the `Wine` class from the well-known Wine-Ontology[9]. Additional properties that are automatically incorporated are for example `hasColour`, `hasMaker`, and `hasFlavour`.

[8]http://wise.vub.ac.be/ontoworld/MetaMapping.owl

[9]http://protege.cim3.net/file/pub/ontologies/wine/wine.owl

3.3.1.5 Domain Mapping (Domain Level)

The *Domain Mapping* defines the mappings from the concepts in the Domain Specification to the concepts in the Virtual Reality Ontology. The purpose of this mapping is to specify how a domain concept (from the high-level problem domain) should by default be represented in the Virtual Environment (by a concept from the low-level target domain). This ontology contains instantiations of ConceptRepresentation and AttributeRepresentation classes defined earlier in the Meta Mapping and therefore can be considered as an instantiation of the Meta Mapping.

The small example below illustrates such a mapping, namely the one called `WineMapping`. This instantiation of `ConceptRepresentation` represents the mapping of the concept `Wine` (source) onto its representable primitive `Bottle` (target). It is given by the `hasSource` and `hasTarget` properties respectively. Next, the attribute of the Wine concept `hasBottleDiameter` is mapped to its equivalent `hasRadius` of the Bottle concept by means of the instantiation of `AttributeRepresentation`. This process needs to be repeated for each of the concepts and obviously for each of the attributes belonging to those concepts.

```
<mm:ConceptRepresentation rdf:ID="WineMapping">
    <mm:hasSource rdf:resource="&ds;Wine"/>
    <mm:hasTarget rdf:resource="&vro;Bottle"/>
    <mm:hasAttributeRepresentation>
        <mm:AttributeRepresentation rdf:ID="hasBottleDiameterTOhasRadius">
            <mm:hasSource rdf:resource="&ds;hasBottleDiameter"/>
            <mm:hasTarget rdf:resource="&vro;hasRadius"/>
        </mm:AttributeRepresentation>
    </mm:hasAttributeRepresentation>
    ...
</mm:ConceptRepresentation>
```

3.3.1.6 World Specification (Instance Level)

The actual specification of the Virtual World, at a conceptual level, is done using the World Specification. This is done in terms of the domain concepts by instantiating concepts defined in the Domain Specification. These instantiations (comparable to instances in Object-Oriented modelling) represent the objects that will eventually populate the Virtual Environment.

The following piece of OWL code illustrates an instantiation of a `Wine` concept called `WiseWine`. First of all, it contains all the properties coming from the fact that it is an instantiation of an already existing class. This tells us for example that the wine has been bottled by `Wise` and has the colour `Red`. Secondly, also the extra properties which were defined need to be given appropriate values. In this example WiseWine is given a bottle-diameter of 7 cm, through an instantiation of the `Diameter` class, with a value of "7" and a unit in "cm" (centimetres).

```
<ds:Wine rdf:ID="WiseWine">
    <vin:hasMaker rdf:resource="#Wise"/>
    <vin:hasColour rdf:resource="&cmo;Red"/>
    ...
    <ds:hasBottleDiameter>
        <cmo:Diameter rdf:ID="BottleDiameter">
            <cmo:hasValue rdf:datatype="&xsd;int">7</cmo:hasValue>
            <cmo:hasUnit rdf:resource="&cmo;cm"/>
        </cmo:Diameter>
    </ds:hasBottleDiameter>
    ...
</ds:Wine>
```

Thus far, only the objects populating the Virtual Environment were defined. In order to use these objects in the Virtual Environment, the designer has to place them into the environment. This can be done either by giving an exact position to an object or alternatively by positioning an object relative to other objects. Suppose `ShelvePL` is defined as an instantiation of the `Shelve` concept from the Domain Specification and has already been located inside the Virtual Enviromnent. Then, the designer can place the bottle of `WiseWine` "**on top of**" the `ShelvePL`. This is represented by creating an instance of the `SpatialRelation` class, defined in the Conceptual Modelling Ontology, as follows:

```
<cmo:SpatialRelation rdf:ID="WiseWineShelvePL">
    <cmo:hasSource rdf:resource="#WiseWine"/>
    <cmo:hasTarget rdf:resource="#ShelvePL"/>
    <cmo:hasType rdf:resource="&cmo;on-top-of"/>
    <cmo:ReferenceFrame rdf:resource="&cmo;FRF"/>
</cmo:SpatialRelation>
```

Next to the fact that an object can be placed inside the Virtual Environment, the designer can also specify relationships that are not directly visible but which are important nonetheless. Suppose, in our department store, there is an area defined as the location of all the red wines called `RedWine`. In the specification below, the bottle of WiseWine is defined as being a part of that particular area. This is expressed by means of the `PlaceArea` type of relation. This is done by making an instance of the `PartWholeRelation` class also defined in the Conceptual Modelling Ontology.

```
<cmo:PartWholeRelation rdf:ID="WiseWineRedWine">
    <cmo:hasSource rdf:resource="#WiseWine"/>
    <cmo:hasTarget rdf:resource="#RedWine"/>
    <cmo:hasType rdf:resource="&cmo;PlaceArea"/>
</cmo:PartWholeRelation>
```

3.3.1.7 World Mapping (Instance Level)

Although instances may be of the same type (concept), they may in some cases require different representations. Therefore, the World Mapping allows (re-)defining the mappings of instances in the World Specification onto concepts in the Virtual Reality Ontology. It thus allows the designer to override the default mappings specified for the concepts in the Domain Mapping, for particular instances.

The code fragment below is quite similar to the one of the Domain Mapping with the exception that now a particular instance is being used as the source, namely `WiseWine`, instead of the concept. The WiseWine is a special brand of wine and therefore it comes with a wooden box instead of only the bottle. It can be seen that the target of the mapping has been changed to a `Box` where it previously was a `Bottle`. As a result, also the mappings of the attributes need to change. The `hasDiameter` property is now being mapped onto the `hasWidth` property. In addition, a rule is used to say that the width (target) is calculated as the diameter (source) with some added value. Note that it is possible to have an m-to-n relationship between the sources and the targets meaning that the same source can be used to map to multiple targets and vice versa. In our case, the `hasDiameter` is being mapped onto the `hasWidth` property but also to the `hasDepth` property (not given here).

```
<mm:ConceptRepresentation rdf:ID="WiseWineMapping">
    <mm:hasSource rdf:resource="&ws;WiseWine"/>
    <mm:hasTarget rdf:resource="&vro;Box"/>
    <mm:hasAttributeRepresentation>
        <mm:AttributeRepresentation rdf:ID="hasDiameterTOhasWidth">
            <mm:hasSource rdf:resource="&cmo;hasDiameter"/>
            <mm:hasTarget rdf:resource="&vro;hasWidth"/>
            <mm:hasRule>
                <mm:Rule>
                    <hasValue rdf:datatype="&xsd;string">
                        &mm;Target = &mm;Source + 2
                    </cmo:hasValue>
                </mm:Rule>
            </mm:hasRule>
        </mm:AttributeRepresentation>
    </mm:hasAttributeRepresentation>
    ...
</mm:ConceptRepresentation>
```

3.3.2 Design Process

Figure 3.3 gives a detailed overview of the different phases that need to be performed together with the ontologies that need to be built within these phases. Since the gap between the conceptual level and the implementation level is too large for it to be bridged in one single step, the design process is divided into three main phases, the specification phase, the mapping phase and the generation phase. Note that the sequential order as given on the figure needs to be preserved but that the process can be iterative. One can do part of the specification, part of the mapping, do the generation and then go back to the specification to complete or correct it until everything is specified.

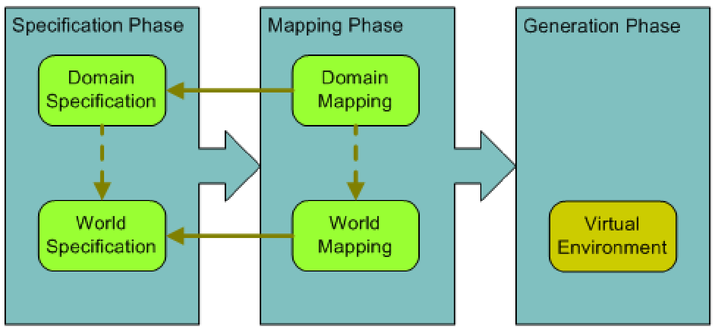


Figure 3.3: *VR-WISE design process*

3.3.2.1 Phase 1: Specification Phase

In the *specification phase*, the designer specifies the Virtual Environment, at a conceptual level, using domain knowledge and without taking any implementation details into account. In principle, this specification phase can completely be performed by the domain expert (or end-user) without the interference of someone familiar with Virtual Reality technology. During this step, the Domain Specification and the World Specification are used. First of all, the designer needs a suitable Domain Specification. This can be done by building one completely from scratch or by selecting, and extending, an existing Domain Specification. Note that it is possible that more than one domain ontology is needed for a single Virtual Environment. For example in the department store, there might be a need for a general Drinks ontology describing other drinks next to wines, and/or a Food ontology describing all edible items in the store. The second part of this phase then consists of building the World Specification by creating instantiations of the concepts in the Domain Specifications for each of the objects one wants to have in the Virtual Environment. In the creation of the specifications, the modelling concepts that are needed are available through the Conceptual Modelling Ontology. This ontology does not need to be created anymore but is provided by the system.

Note that, in general, it is too laborious and not very user-friendly to ask the designer to write OWL specifications. Instead of requiring background knowledge in Virtual Reality technology, this would require knowledge about ontologies and the OWL language in particular. In most cases, this also does not belong to the general background knowledge of a domain expert. So, an additional layer of abstraction has been built on top of the ontology layer, namely an intuitive graphical notation. For the different high-level modelling concepts defined in the Conceptual Modelling Ontology, a suitable graphical representation has been developed. In fact, the designer now has the choice of modelling the Virtual Environment using the ontologies (usually for more experienced people) or by means of drawing intuitive diagrams. More details about this graphical notation will be given in the following chapters.

3.3.2.2 Phase 2: Mapping Phase

After the specification phase, the *mapping phase* involves specifying how the concepts and the instances need to be represented in the Virtual Environment. This is done by specifying the mappings for the concepts defined in the Domain Specification and the instances defined in the World Specification (both from the conceptual level) towards the Virtual Reality Ontology. This step uses the Domain Mapping and the World Mapping. First, in the Domain Mapping, the default mappings are specified. Automatically, all the instances in the World Specification are mapped according to their corresponding type-level mappings. Next, the World Mapping provides the opportunity to override any of these default mappings. This step is optional. Since in this phase, the designer comes in close contact with implementation issues, there is usually a tighter cooperation needed with the technology expert that may assist the designer in defining the appropriate mappings.

3.3.2.3 Phase 3: Generation Phase

The *generation phase* generates the actual source code for the Virtual Environment specified in the specification step using the mappings defined in the mapping step, i.e. the conceptual specifications given by means of the Domain Specification and the World Specification are converted into a working application by means of the conceptual mappings given by the Domain Mapping and the World Mapping. This phase is entirely supported by means of a dedicated software tool called *OntoWorld*. The tools supporting the VR-WISE approach will be discussed in more detail in chapter 7. In short, the generation process goes as follows. All the instances in the World Specification are taken one by one. The World Mapping is searched for any mapping having the instance in question as a source. If this is the case, the instance is mapped according to this object-level mapping. Otherwise, its corresponding type is retrieved and the instance is mapped according to the type-level mapping specified in the Domain Mapping. This will give us an implementation-level specification that can be easily converted into a particular Virtual Reality language. At this moment, only X3D code is generated for the Virtual Environment. However, the approach is developed in such a way that a switch between different target languages can be easily accomplished. Figure 3.4 illustrates the final Virtual Environment resulting from the generated code for the department store.

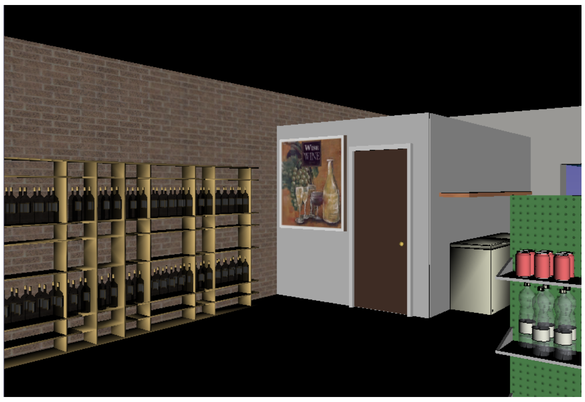


Figure 3.4: *The resulting Virtual Environment*

3.4 Extension of VR-WISE

In the approach as described in this chapter, nothing was mentioned about the modelling of behaviour. At the beginning of the VR-WISE research, the behaviour modelling capabilities were not yet available. It is the aim of this dissertation to extend (or build upon) the VR-WISE approach to also enable the specification of the dynamical aspects with the goal to automatically generate programming code from it. Basically, the work can be situated at the meta-level. New conceptual modelling concepts need to be devised for modelling behaviour, and the Conceptual Modelling Ontology will be extended with a dedicated Behaviour Modelling Ontology at the meta-level containing these high-level modelling concepts for the specification of behaviour in Virtual Environments. This extension mainly involves the first phase of the design approach (specification phase) and implicates that the Domain Specification as well as the World Specification will then be able to make use of these concepts in the specifications of the Virtual Environment. Chapters 4, 5 and 6 will give more details on this.

3.5 Summary

In this chapter, the VR-WISE approach for designing Virtual Reality applications has been presented. This approach is used as the basic framework for the research performed for this dissertation. The idea underpinning this approach is that the design of a Virtual Environment should start at a conceptual level without taking into account any implementation details whatsoever. The ultimate goal of the VR-WISE research is to facilitate and shorten the development process of VR applications by means of conceptual specifications.

It is our belief that the introduction of a conceptual modelling phase into the overall development process of Virtual Environments, will facilitate the use of Virtual Reality and make it available to a much broader public including non-VR-experts. In fact, the current situation in Virtual Reality is quite similar to the past situation in the software engineering and information systems domains. Here, the uses of a conceptual modelling language have proven to be of much benefit. However, the modelling languages used in these domains are not appropriate for the design of Virtual Environments.

Furthermore, VR-WISE uses the concept of ontologies as a mechanism to specify the conceptual models. Ontologies provide us with a mechanism, understandable by both man and machine, to capture semantically rich information about the Virtual Environment to be built, which may also be exploited when the Virtual Environment is used. As mentioned in this chapter, ontologies also provide us with a number of advantages such as the improved communication, inter-operability and reusability. Furthermore, they facilitate consistency checking, searching and querying.

The starting point for the VR-WISE approach is the use of intuitive high-level specifications of the Virtual Environment one wants to realize. Due to the use of domain terminology and intuitive modelling concepts, there is a strong similarity between how one describes the Virtual Environment in our approach and how it would be done using natural language. Therefore, no background knowledge in Virtual Reality enabling technologies is required for the designer to model a Virtual Environment using the VR-WISE approach. Furthermore, all modelling concepts provided to the designer are carefully chosen, watching over the general characteristics required for conceptual modelling languages. To convert the high-level specifications into a working application, in a next step, a mapping from the conceptual level into the implementation level is specified. Finally, the actual source code for the Virtual Environment is (semi-)automatically generated from the high-level specifications using these mappings.

CHAPTER 4

Conceptual Modelling of Dynamic Virtual Environments

In the previous chapter, the methodology used in this dissertation was explained. It started by introducing the meaning of the term *conceptual modelling*. It then described the use of ontologies both as a way of formalizing the high-level modelling concepts and as a way to model a Virtual Reality application. It also introduced our own high-level design method for Virtual Environments called VR-WISE, based on ontologies. This chapter will now continue to build further on the general VR-WISE approach and lay the foundations for the modelling of dynamic Virtual Environments, which will be treated in the next two chapters of this dissertation.

The remainder of this chapter is structured as follows. Section 4.1 presents some general characteristics of our behaviour modelling approach together with the requirements to be taken into account when modelling behaviour in Virtual Environments. Section 4.2 will then give an informal description of the behaviour modelling approach developed. Section 4.3 discusses in details how the ontology-architecture presented in previous chapter is extended to integrate the high-level modelling concepts used for the modelling of behaviour. It also describes the ontologies added for the specification of the behaviour. Furthermore, it also discusses the approach used for behaviour modelling. Section 4.4 describes how the behaviour modelling fits into the already existing VR-WISE development process. Section 4.5 gives a summary and a discussion about what has been presented in this chapter.

4.1 Initial Requirements

The behaviour modelling approach that is presented in this dissertation aims at providing content experts (domain experts, usually not experts in VR) with a means to design object-behaviour at a high-level and at the same time allowing more experienced designers to keep control over the granularity of these specifications. Here, some general characteristics of the behaviour modelling approach that is put forward, and requirements for this behaviour modelling approach, are raised.

As mentioned earlier in the introduction and in the chapter on related work, the majority of designers still takes a code-only approach, at least on the part of specifying the behaviours, and do not use a high-level modelling approach for designing behaviours. They rely almost entirely on the code they write and the behaviours are directly programmed in some kind of dedicated scripting language. The behaviour modelling approach presented here is a **model-based** approach, meaning that high-level models are used to describe the behaviour at a conceptual level. In addition, the models have sufficient details to enable the generation of a full implementation. By using conceptual models, the specifications are made at a much higher level than scripting languages. Furthermore, if any scripting is needed, it should be done in a declarative way so that it still remains accessible to non-programmers.

Most model-based approaches encountered in the related work often used (or extended) a finite state machine as the underlying model for describing the behaviour of objects. They are based on the different states the object can be in during the simulation. Our behaviour modelling approach follows a different path, which we call **action-oriented**. This approach is focusing on the different actions that an object may take throughout the simulation rather than on the states an object can be in. It is our belief that using actions is more natural than using states for specifying the behaviour in a Virtual Environment.

A traditional animation process uses transformation operations for modifying the numerical data describing the objects in space. These transformation operations are represented by matrices or quaternions. Although modelling tools allow manipulating these transformations through more friendly user-interfaces, it still requires some background knowledge in mathematics to correctly create such transformations. Our approach tries to provide a higher level of abstraction and use more **intuitive actions**. The transformation operations can then be replaced by these intuitive actions where they are equally powerful. Actions can be structured in a **hierarchical** way which improves the scalability, modularity and reusability of the behaviour specifications.

An animation usually consists of a number of transformations that need to be set correctly throughout time. These transformations are usually not only sequential but are often related to each other in a more complex way. One of the most difficult aspects for creating compelling behaviours (animations) is this time setting of the actions within the complete simulation. The approach described in this dissertation provides modelling concepts that allow **richer and intuitive timings** of the actions. Through the use of so-called temporal relations, the ordering in time of the actions can be done in more different ways than only sequentially.

Transformations can occur on single objects, on a collection of objects or on the entire environment. Thus, somehow, these should be connected to each other. Furthermore, they can occur as a reaction to different events. The modelling approach should facilitate a loose coupling between the behaviour specification and the link to the objects on the one hand and the connection with the events that trigger the behaviours on the other hand. This is needed to reduce the complexity and to enhance reusability. The proposed approach allows us to define different aspects independently and then in a subsequent step, link the different aspects together to form the final result. To achieve this independence, the behaviour modelling approach presented here is divided into two separate steps (see following section).

4.2 Overview of the Behaviour Modelling Approach

The behaviour modelling approach that is introduced here, is developed with the issues and requirements stated above in mind. The approach consists of two sequential steps namely the **Behaviour Definition** and the **Behaviour Invocation** [Pellens et al., 2007].

The first step, the behaviour definition, allows the designer to define the different behaviours for an object. Note that in our approach, the behaviours of an object are defined separated from the specification of the visual appearance of the object and independent of how the behaviour will be triggered. This improves reusability and enhances flexibility as the same behaviour definition can be reused for different objects (if different types of objects have the same behaviour) and/or can be triggered in different ways (e.g., by some user interaction or by a collision with another object). A behaviour definition is specified using a so-called *Behaviour Definition* model. For the Behaviour Definition model, the most important modelling concept is the actor. An *actor* is a placeholder for an object. A *behaviour* will be defined for an actor and later (in the second step, the behaviour invocation) the behaviour is assigned to an actual object. This approach has been taken to separate the specification of a behaviour from the specification of the visual appearance of the object that will own the behaviour. An additional advantage is that the same behaviour can be assigned to different (types of) objects. A number of primitive actions (behaviours) have been defined in our approach ranging from basic manipulations (e.g., move, roll, turn, resize,...) up to actions capable of restructuring the scene graph (e.g., construct, destruct, disperse, group,...). The primitive actions and defined behaviours can be combined through a set of *operators* to form composite behaviours. Temporal operators (e.g., before, meets, overlaps,...) can be used for synchronization purposes; lifetime operators (e.g., enable, disable,...) are used to control the lifetime of behaviours; and conditional operators are used to control the flow of the specification.

In the second step, the behaviour invocation, the behaviours that were defined in the behaviour definitions are assigned to the actual objects in the Virtual Environment. Furthermore, in these models the mechanism of how the behaviours assigned to objects may be invoked, i.e. the events that may trigger them, is denoted. In this way, the invocation of the behaviour is also separated from the actual definition of the behaviour. Hence, the same behaviour can be triggered in different ways depending on the context in which it is used. The behaviour invocation is specified using a so-called *Behaviour Invocation* model. The most important modelling concept for the Behaviour Invocation model is the *object* representing the actual objects populating the Virtual Environment. By assigning an actor to an object, the object receives all the behaviours defined for this actor. The objects used in these models refer to concepts or instances in the models defining the static structure of the Virtual Environment (Domain Specification and World Specification). In this way, the link with the models describing the static structure is established. The *behaviour reference* concepts used in the Behaviour Invocation models are actually instantiations of the behaviours defined in the Behaviour Definition models. This provides the link between the behaviour invocation and the behaviour definition. Furthermore, the modelling concept *event* is used to describe when the behaviours should be triggered. This is done by attaching events to behaviours. There are a number of possible events, namely time events for a triggering based on the passing of time, user events to trigger behaviour as a reaction to user interaction, context events to react to changes from the environment, and object events to invoke behaviour upon objects interacting with each other.

In the same line as with the overall VR-WISE approach, the specification of the conceptual models for the object behaviour in Virtual Environments is supported by means of ontologies. Therefore, the architecture of ontologies for the VR-WISE approach had to be extended. This will be discussed in more details in the following sections. Then, in the following chapter, the graphical behaviour modelling language developed for the approach is discussed. This graphical language allows the designer to specify the different models in a graphical way.

4.3 Extended Architecture of Ontologies

The ontology-architecture mentioned in chapter 3 contains the ontologies that were used at different levels of abstraction in order to describe the Virtual Environment. However, at the start of this PhD work, there were no modelling concepts available to allow the designer to model the behaviour of the objects in a Virtual Environment. Therefore, the architecture of ontologies (see figure 3.2 on page 63) has been extended to incorporate the modelling concepts that allow the designer to specify the behaviours of the objects in a Virtual Environment. The new modelling concepts provided are consistent with the approach taken by VR-WISE, i.e. modelling must be done from a high-level point of view and independent of any technological issues. The extended architecture is shown in figure 4.1. The architecture was extended at two different levels, namely on the "Meta Level" the Behaviour Modelling Ontology has been added, and on the "Domain Level" the Behaviour Specification has been added. The extensions are highlighted in bold.

- On the Meta Level, the Behaviour Modelling Ontology has been added as part of the general Conceptual Modelling Ontology. This new ontology defines the high-level modelling concepts for the behaviour specifications.

- The Domain Level has been extended with the Behaviour Specification. This ontology is actually a combination of two ontologies, the Behaviour Definition Ontology (domain-dependent but application-independent) and the Behaviour Invocation Ontology (application-dependent) and gives the specification of the behaviours of a VE.

Looking at figure 4.1, the reader may see that no additional mapping ontology has been added for the behaviour modelling. This is not needed. For the static structure, the designer needs to define the concepts from the application domain needed in the Virtual Environment. These concepts are defined independent from their visual representation in the Virtual Environment. Later on in the development process, the designer specifies how the objects must be represented in the Virtual Environment by mapping these domain concepts onto a particular VR representation. Therefore, mapping ontologies are needed. However, for the modelling of behaviour only high-level predefined modelling concepts are used. In other words, the designer cannot define his own modelling concepts[1]. Because the modelling concepts are all predefined, the mappings towards an implementation are predefined as well, i.e. the behaviour specifications are automatically mapped to code extractions. A second reason why there is no explicit mapping ontology

[1]In the design pattern framework extending the behaviour modelling approach, the designer *is* allowed to create his own modelling concepts as will be discussed in chapter 6.

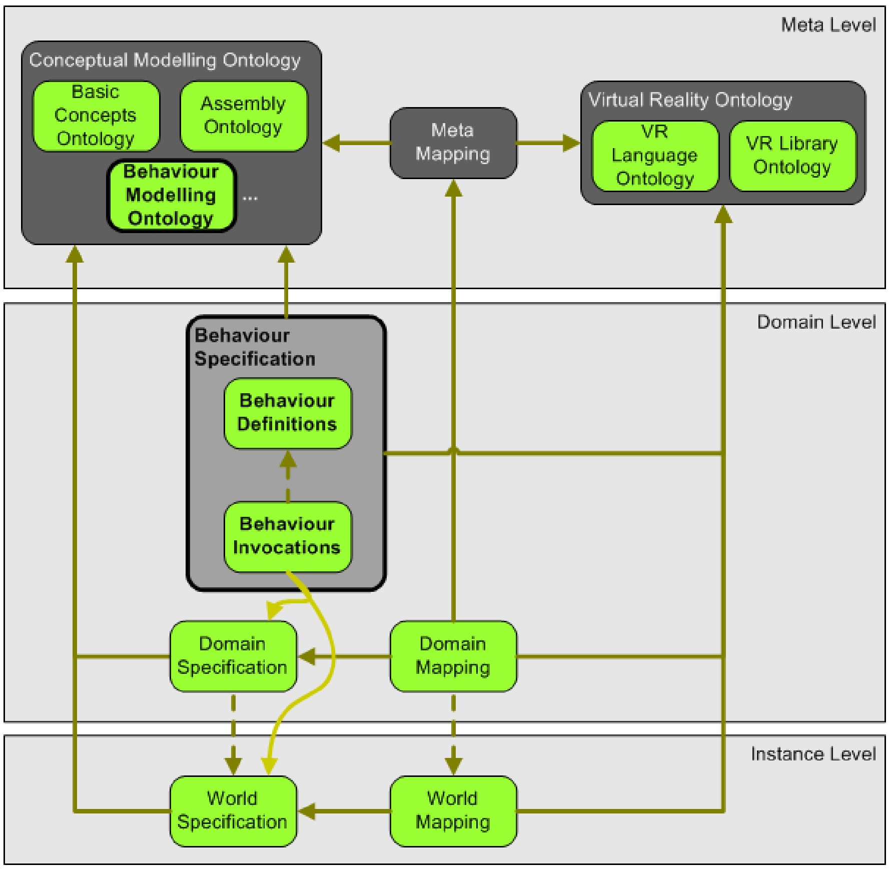


Figure 4.1: *Extended architecture of ontologies*

for behaviour, is the fact that the high-level modelling concepts need to be mapped onto source code extractions expressed in a programming language. The structure of the source code is influenced by the way in which the behaviour is processed by the target technology. It is often dependent on the frame-by-frame execution of the rendering engine. In other words, the same code is executed each rendered frame for as long as the behaviour is active. This results in programming code that is often not just a sequence of instructions but is structured in a more complex manner. In principle, it would be possible to specify this by means of a mapping but it would not be as straightforward as in the case of the modelling concepts for the static structure. In addition, the added value of such a mapping would not set off the effort of specifying it. For these reasons, we have opted not to bother the designer with manually mapping the behaviours but instead provide default mappings to some implementation technology. It is always possible to add new mappings for different implementation technologies.

The remaining part of this section explains in more detail the ontologies that were developed to extend the existing ontology architecture of the VR-WISE approach. Again, the ontologies are built using the Protégé-OWL ontology editor.

4.3.1 Behaviour Modelling Ontology (Meta Level)

At the meta-level, a new ontology has been developed, called the *Behaviour Modelling Ontology* (BMO) [Pellens et al., 2005c]. This ontology is part of the general *Conceptual Modelling Ontology* (see figure 4.1). It contains the different high-level modelling concepts that may be used to create a Behaviour Definition model and a Behaviour Invocation model (see next section). It also describes their properties and relationships between each other. This ontology serves as a meta-model, namely it formalizes what makes a valid behaviour specification. In our case, an ontology is used for the meta-model because it fits perfectly with the rest of the VR-WISE approach. However, we could have used another specification formalism instead. A first advantage of having an explicit meta-level is that through the meta-model, the different modelling concepts are formally defined. Secondly, having the meta-level, the Behaviour Modelling Language can be easily extended to incorporate new concepts or features. A comparable situation exists for UML where also a meta-level is provided and for which a number of extensions have been developed such as STUML (Spatio-Temporal UML) [Price et al., 1999] or OMMMA (Object-oriented Modelling of Multimedia Applications) [Sauer and Engels, 2001]. Also note that extending scripting languages is not that easy. This section will now discuss how the different modelling concepts for designing the behaviour, are defined within our Behaviour Modelling Ontology.

The concepts in the Behaviour Modelling Ontology can be divided into two parts. Figures 4.2 and 4.3 give graphical depictions of these two parts. Due to space restriction, these figures do not display the complete ontology. The reader is referred to the appendix for the complete ontology specified in OWL (see appendix A). Each of the two figures corresponds to one of the steps of the behaviour modelling approach, namely the behaviour definition and the behaviour invocation. They will be discussed in more detail in the following section. For the graphical representation, the meta-models, defining the different modelling concepts of the behaviour modelling approach are specified as UML class diagrams in order to clarify the contents of the Behaviour Modelling Ontology.

Figure 4.2 shows an extraction of the meta-model of the Behaviour Definition model. Note that all the concepts in this diagram are in fact subclasses of the more general `BehaviourDefinitionThing`. In this way, we express that these are the modelling concepts that can be used in the behaviour definition.

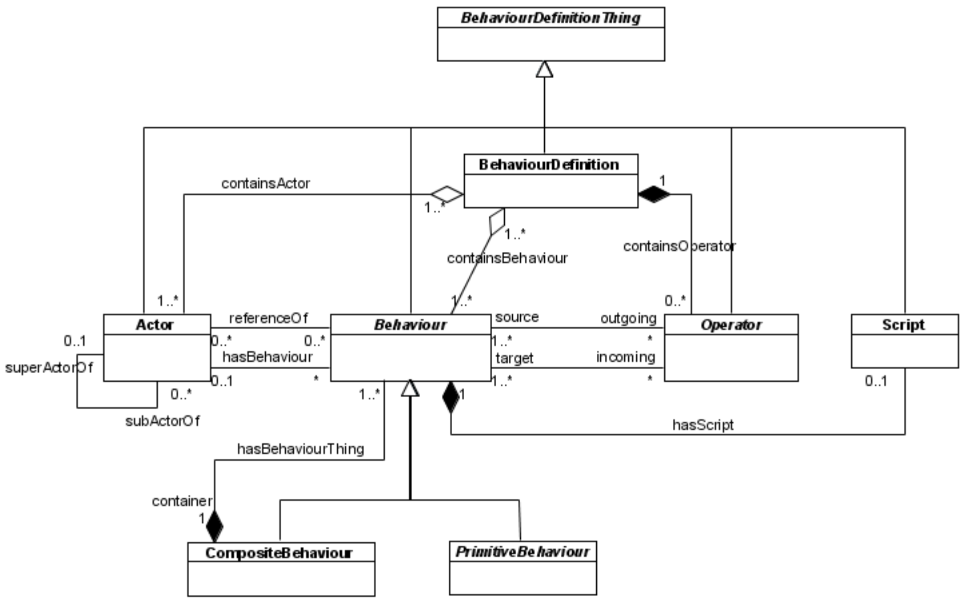


Figure 4.2: *Meta-model of a Behaviour Definition Model*

The BehaviourDefinition concept is the main concept which actually represents the model that specifies a behaviour definition. Creating a new behaviour definition implies creating an instance of this concept.

A Behaviour Definition model can contain one or more Actor concepts as specified by the containsActor relation. An actor has to be part of at least one behaviour definition but can be part of more than one behaviour definition. Actors are representing objects that are involved in a behaviour. Actors can also be part of a generalization/specialization hierarchy. This is given by the superActorOf-subActorOf relation stating that a super-actor can have zero or more sub-actors while a sub-actor has at most one super-actor. Furthermore, they may contain instantiations of the Property concept (not included in the figure). This concept is imported from the *Basic Concepts Ontology* and allows to define properties for the actor. The actors may be associated with the behaviours in two ways. Firstly, actors may have multiple behaviours defined for them as shown by the hasBehaviour relation. Secondly, a behaviour may need some actors as reference (further explained in section 5.3.3). This is expressed by the referenceOf relation.

A Behaviour Definition model also contains a number of Behaviour concepts that represent the actual actions (behaviour) that an object (represented by an actor) is able to undertake. This link between a behaviour and an actor is represented by the containsBehaviour relation. The Behaviour concept is an abstract concept, it can only be instantiated through one of its sub-concepts, namely a PrimitiveBehaviour or a CompositeBehaviour. A number of primitive behaviours have been defined in our approach ranging from basic manipulations (e.g., move, turn, roll,. . .) up to actions capable of restructuring the scene graph (e.g., construct, destruct, group, disperse,. . .). A composite behaviour consists of at least one sub-behaviour, which can on its turn either be a primitive or composite behaviour, as given by the `hasBehaviourThing` relationship. A composite behaviour can be named, in which case it can be referenced more than once in other behaviour definitions. Such a behaviour is called a `SubBehaviour`. It is actually a special kind of composite behaviour and is therefore defined as sub-concept of `CompositeBehaviour`. A `SubBehaviour` facilitates reuse and modularity of the specifications. A behaviour may have a `Script` concept attached to it, as expressed by the `hasScript` relation.

Finally, a Behaviour Definition model contains a number of `Operator` concepts, represented through the `containsOperator` relation. The purpose is to link a number of behaviours to each other to create composite behaviours. The operators form a direct relation between one or more `incoming` behaviour (called `source` of the operator) and one or more `outgoing` behaviours (called `target` of the operator). The `Operator` concept is also an abstract concept meaning that it cannot be instantiated directly but only its sub-concepts can be instantiated. Different kinds of operators are supported by the approach. An operator can belong to only one behaviour definition.

The second part is shown by figure 4.3 and depicts an extraction of the meta-model of the Behaviour Invocation model. Similar as for the meta-model of the Behaviour Definition model, the modelling concepts for this model are sub-concepts of the abstract concept, `BehaviourInvocationThing`. The classes depicted in red are defined in the previous figure (so in the same ontology), or in a different ontology (e.g., Basic Concepts Ontology).

A Behaviour Invocation model consists of a number of `Object` concepts as shown by the `containsObject` relationship. These objects represent the actual objects that are populating the Virtual Environment. They refer to either a `Concept` or an `Instance`, which have been defined in the *Basic Concepts Ontology*. They provide the link between the behaviour specification and the static structure specification of the Virtual Environment. Note that a single concept or instance can be used in multiple behaviour invocations. Furthermore, an object is associated to at least one `Actor` as specified by the `playsRoleOf` relation. By linking an actor to an object, it will receive all the behaviours defined for this actor. The other way around, an actor can be assigned to multiple objects.

A Behaviour Invocation model also contains at least one `BehaviourReference` concept as shown by the `containsReference` relationship. A behaviour reference is associated with exactly one `Behaviour` concept defined in a Behaviour Definition model. The other way around, a behaviour can be referenced by multiple behaviour references, i.e. it can be used in different Behaviour Invocation models. This is represented by the `hasReference`

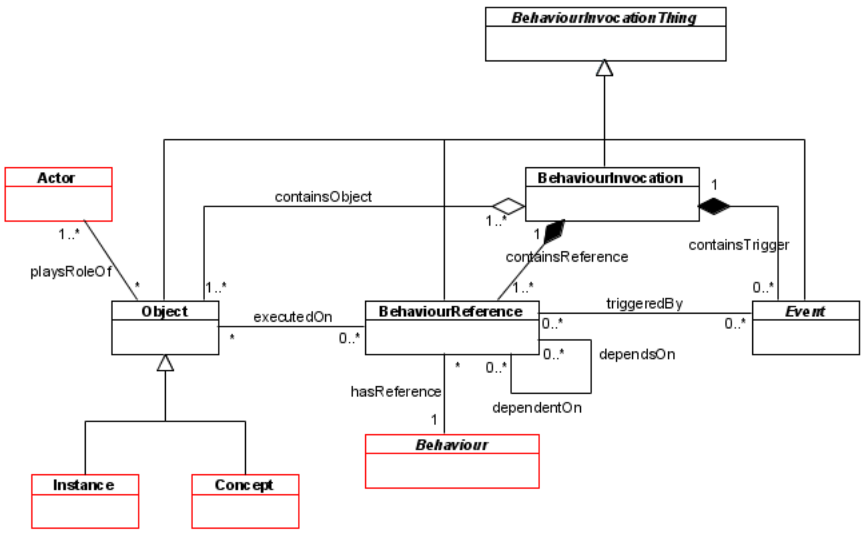


Figure 4.3: *Meta-model Behaviour Invocation Model*

relation. In the same way as the behaviour and the actor were related to each other, the behaviour reference and the object are also related to each other through the `executedOn` relation. Furthermore, instantiations of a behaviour reference are related to each other through dependencies denoting a particular causal link between them as given by the `dependentOn-dependsOn` relation. It says that zero or more behaviour references can depend on (i.e. call or use) another behaviour reference. The other way around, one behaviour reference can be dependent on (i.e. called by or used by) zero or more other behaviour references.

A third main modelling concept for the Behaviour Invocation model is the `Event` concept which is expressed by the `containsTrigger` relation. There can be zero or more events for one Behaviour Invocation model. A single event can trigger multiple behaviours but a single behaviour can also be triggered by multiple events. This is expressed by associating an event with a behaviour through the `triggeredBy` relation. The `Event` concept is also an abstract concept meaning that it can only be instantiated through one of its sub-concepts. Different kinds of events are available to describe different means of triggering a behaviour.

Next to the Basic Concepts Ontology, the Behaviour Modelling Ontology also imports statements from the well known Time Ontology[2] [Hobbs and Pan, 2004]. This is an ontology that has been developed to describe temporal information and was originally developed in the context of the Semantic Web. It covers measures of duration, meanings of clock and calendar terms and so on. The concepts `CalendarClockDescription` and

[2]http://www.isi.edu/ pan/damltime/time-entry.owl

DurationDescription for example are used within the specifications of the behaviour. Furthermore, the terms defined within this ontology can also be exploited to make it possible to reason about the behaviour specifications [Hobbs and Pustejovsky, 2003].

4.3.2 Behaviour Specification (Domain Level)

At the domain level, the ontology-architecture was extended with the Behaviour Specification (see figure 4.1). As briefly discussed before, the Behaviour Specification, i.e. the process of creating the behavioural specifications, is broken down into two separate steps namely the definition and the invocation resulting in two different and separate models called the *Behaviour Definition* model and the *Behaviour Invocation* model.

4.3.2.1 Behaviour Definition Model

The first step of the behaviour modelling process consists of setting up the Behaviour Definition model. A behaviour definition is used to specify the different behaviours for an object. This is done by instantiating the subclasses of BehaviourDefinitionThing discussed above.

The example that follows shows an extraction of a Behaviour Definition model (in OWL format). This specification contains an actor called `Door` for which a behaviour has been defined called `OpenDoor`. This is done through the `hasBehaviour` relation.

```
<bmo:BehaviourDefinition rdf:ID="OpenDoorDefinition">
    <bmo:containsActor>
        <bmo:Actor rdf:ID="Door">
            <bmo:hasBehaviour rdf:resource="#OpenDoor"/>
            ...
        </bmo:Actor>
    </bmo:containsActor>
    <bmo:containsBehaviour>
        <bmo:CompositeBehaviour rdf:ID="OpenDoor">
            ...
        </bmo:CompositeBehaviour>
    </bmo:containsBehaviour>
</bmo:BehaviourDefinition>
```

4.3.2.2 Behaviour Invocation Model

After the definitions have been created, the second step involves the creation of a Behaviour Invocation model for each of the Behaviour Definition models. This is achieved by instantiating subclasses of BehaviourInvocationThing. Remember that the definition of behaviour is done independently from the actual objects in the Virtual Environment. To connect behaviour to actual objects, this second step is needed. A Behaviour Invocation model assigns the behaviours defined in a Behaviour Definition model to actual objects. It also denotes how the behaviours may be invoked, i.e. the events that may trigger the behaviours of the objects. In this way, also the invocation is separated from the actual definition of the behaviour. Hence, the same behaviour can be triggered in different ways depending on the situation.

The example that follows shows an extraction of a Behaviour Invocation model (also in OWL format). This specification contains an object called `Entrance`, a behaviour reference called `OpenDoorReference` and an event called `Timer`. The object refers to a concept called `Gate` with the `refersTo` relation. In addition, it plays the role of the `Door` actor defined previously, and given by the `playsRoleOf` property. The behaviour reference refers to the `OpenDoor` behaviour by means of the `hasReference` property. It is performed on the `Entrance` object and is triggered by the event `Timer`, defined by the `executedOn` and `triggeredBy` relations respectively.

```
<bmo:BehaviourInvocation rdf:ID="OpenDoorInvocation">
    <bmo:containsObject>
        <bmo:Object rdf:ID="Entrance">
            <bmo:refersTo rdf:resource="&ds;Gate"/>
            <bmo:playsRoleOf rdf:resource="&bd;Door"/>
            ...
        </bmo:Object>
    </bmo:containsObject>
    <bmo:containsReference>
        <bmo:BehaviourReference rdf:ID="OpenDoorReference">
            <bmo:hasReference rdf:resource="&bd;OpenDoor"/>
            <bmo:executedOn rdf:resource="#Entrance"/>
            <bmo:triggeredBy rdf:resouce="#Timer"/>
        </bmo:BehaviourReference>
    </bmo:containsReference>
    <bmo:containsTrigger>
        <bmo:OnTimeEvent rdf:ID="Timer">
            ...
        </bmo:OnTimeEvent>
    </bmo:containsTrigger>
</bmo:BehaviourInvocation>
```

4.4 Fitting Behaviour Modelling into the Overall Development Process

The use of the ontologies in the VR-WISE approach have gone hand in hand with a development process consisting of three phases, namely the specification phase, the mapping phase, and the generation phase. The Behaviour Specification is a sub-phase of the specification phase of the development process. It is a part of the specification of the Virtual Environment in the same way as the Domain Specification and World Specification are.

In the first step, the designer needs to specify a Behaviour Definition. The Behaviour Definition can actually be done at any moment in time during the specification phase. It can effectively be created before the Domain Specification or the World Specification is made, as well as after the complete specification of the static structure. This is because the definitions of the behaviours are completely independent of the static structure of the (objects in the) Virtual Environment. The second step consists of creating the Behaviour Invocations. Here, the definitions of the behaviour are 'instantiated' and parameterized for the particular context. Therefore, this step always needs to be performed after the first step of the behaviour modelling approach. Furthermore, also connections with the static structure of the Virtual Environment are needed, therefore this step must be done after the Domain Specification (when the behaviours can be attached to concepts) or after the World Specification (when the behaviours need to be attached to the instances of the concepts).

Obviously, both steps, namely the Behaviour Definition and the Behaviour Invocation, need to be finished before the actual generation phase can be performed. When the source code for an object in the Virtual Environment is generated, the system retrieves any possible behaviour that might have been attached to this object (or to its concept type) by means of the Behaviour Invocation models. If it has found behaviours, the appropriate source code extractions will be assembled on the basis of the definitions in the Behaviour Definition models (if not already generated before). Then, the initialization code for this particular object is generated and merged with the rest of the code. This process is repeated for each of the objects.

In many cases, the specification of the behaviour will be done in an iterative process together with the specification of the static structure where first some objects are specified, and then their initial behaviours are defined after which code is generated to inspect the result and the designer goes back to further refine the objects or to add new ones.

4.5 Summary

In this chapter, the principles of our behaviour modelling approach were discussed together with its integration into the general VR-WISE approach described in chapter 3.

The approach that is proposed in this dissertation tries to overcome some of the difficulties encountered with specifying behaviour in classical approaches. Our approach is a model-based approach meaning that the behaviour is specified through high-level models. In our approach, these high-level models are specified through the use of ontologies. This is in contrast with the more traditional code-based approaches where the behaviours of Virtual Environments are described through source code in a scripting language or by using some general purpose programming language.

The behaviour modelling approach consists of two steps. In the first step, called the Behaviour Definition, behaviours are defined providing only a loose coupling with the objects for which they are defined and independent of the events that will trigger them. In the second step, namely the Behaviour Invocation, the behaviours are actually connected to the objects and the events that may trigger them. Finally, in the generation step of the overall VR-WISE approach, the pieces of source code that are generated for the dynamic aspects are merged together with the rest of the source code (generated for the static aspects) to come to the final result, a dynamic Virtual Environment.

The behaviour modelling language in our approach includes a number of modelling concepts, which are described in the Behaviour Modelling Ontology. Furthermore, the Behaviour Modelling Ontology also prescribes *how* the high-level modelling concepts can be used in a specification. The Behaviour Modelling Ontology actually extends the Conceptual Modelling Ontology located at the meta-level of the ontology architecture of the VR-WISE approach. For all the modelling concepts that are defined in the Behaviour Modelling Ontology, graphical representations have been developed, allowing the designer to visually construct the behaviour specifications. The following chapter will discuss the different modelling concepts in more detail as well as their graphical notation and their semantics.

CHAPTER 5

Graphical Behaviour Modelling Language

In the previous chapter, the principles behind the behaviour modelling approach were discussed. It was explained how the general VR-WISE architecture was extended with the so-called Behaviour Modelling Ontology. A specification of the overall structure of the ontology was also given. Furthermore, the different steps involved in the process of modelling behaviour were discussed. In this chapter, the different behaviour modelling concepts are described in more detail. Both the definitions (an informal as well as a more formal) and the notation for the visual representation of the different high-level behaviour modelling concepts are given.

We arrived at the modelling concepts presented here following a top-down approach. A number of high-level modelling languages such as UML, ER and ORM were investigated for useful modelling concepts that could be reused in the context of modelling object behaviour in Virtual Environments. From these modelling languages, a number of general modelling concepts were identified to be useful. Furthermore, three different domains were investigated as well, namely the geographical domain, the architectural domain, the design and manufacturing domain, and the medical domain. We mainly focussed on these domains as they have been using VR for a long time and as they have also some well established high-level concepts which might be appropriate for VR. The useful modelling concepts found in these domains were generalized to be applicable to VR in general. Furthermore, the requirement to have an intuitive and easy to understand set of modeling concepts has been a major decision criterion in the research approach followed.

Please note that the more formal definitions do not capture the complete semantics of the behaviour modelling concepts. The focus here is on their syntax. Within this chapter, a number of conventions are used:

- Constant, predicate, and function symbols are capitalized.
- Sets are introduced by giving the singular name (or an abbreviation) of the required type, in capitals and in bold.
- Variables and set elements start with a lowercase letter.

Also a few commonly used sets often returning are:

- $\emptyset$ denotes the empty set,
- $\mathbb{S}$ denotes the set of strings,
- $\mathbb{R}$ denotes the set of real numbers, and
- $\mathbb{N}$ denotes the set of natural numbers.

The chapter is structured in the following way. First, in section 5.1, some basic modelling concepts used in the rest of this chapter are clarified. Section 5.2 explains the concept of Structure Chunk. This is a small diagram type, which is needed later in the chapter[1]. Afterwards, the two main diagram types for modelling behaviour are discussed, namely the Behaviour Definition Diagram in section 5.3 and the Behaviour Invocation Diagram in section 5.4. These are the diagram types used for the two corresponding steps, the behaviour definition and behaviour invocation, explained in the previous chapter. For the modelling concepts discussed in these diagrams, the following structure is used: an informal explanation is given about the concept itself, followed by a more formal definition and afterwards the graphical notation is introduced. In section 5.5, the scripting language, part of our mixed graphical/textual modelling language, is introduced. Finally, section 5.6 will give a short discussion about this chapter.

5.1 Basic Concepts

Before going into the different types of diagrams that are supported by our approach, it is worth clarifying some basic concepts about modelling 3D worlds or Virtual Environments. These concepts are very important since many of our modelling primitives heavily rely on them. Reading this section will give a better understanding of what is going to follow in the rest of this chapter.

A first issue which needs to be explained is the concept of *Space*. In order to model a 3D world or Virtual Environment, the three-dimensional

[1]The reader might want to skip this section for now and come back to it later when it is actually used in the main diagram types.

or virtual space has to be defined properly. This space has, as the name suggests, three dimensions namely the width, the depth and height. These dimensions are represented by axes. In this work, the convention is taken that the axes are labelled with x for the width dimension, y for the height dimension and z for the depth dimension. All three axes are mutually perpendicular. The point in which these axes cross each other is the origin of the space. A location (or position) within the space can now be defined by means of a coordinate on the three axes that make up the space (`(x,y,z)` coordinates). Figure 5.1a shows an example of the space as it is defined here. The coordinate systems that are used in this dissertation are always right-handed coordinate systems meaning that values on the x-axis become larger to the right of the origin, on the y-axis they become larger above the origin and on the z-axis they grow as they move to the front.

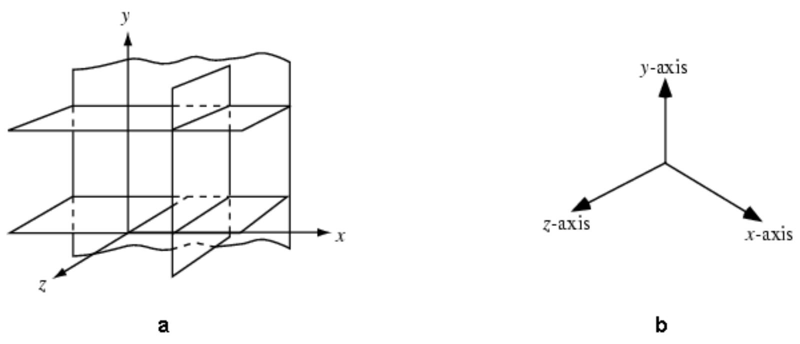


Figure 5.1: *Three dimensional space*

After having defined the space, one extra concept needs to be clarified, namely a *Reference Frame*. One way to think about a reference frame is as a three-dimensional coordinate system attached to the object for which the position or movement can be specified [Frank, 1998]. In most modelling packages, a reference frame is defined as having an origin, usually at the centre of the object (e.g., coordinate `(0,0,0)`). Next to the origin, a reference frame also has three axes, the x-axis, the y-axis and the z-axis (see figure 5.1b). Finally, the reference frame is further defined by means of a handedness describing the way the axes are related to each other. In a right-handed reference frame, the positive x- and y-axes point right and up, and the negative z-axis points forward. In a lef-handed reference frame, the positive x-, y- and z-axes point right, up and forward, respectively.

Definition 1 (Reference Frame). *A reference frame* `RF` *is defined as a pair* $\langle$*Location, Orientation*$\rangle$ *where:*

- *Location* $\in \mathbb{R} \times \mathbb{R} \times \mathbb{R}$ *is the coordinate of the origin,*

- *Orientation* $\in \mathbb{R} \times \mathbb{R} \times \mathbb{R}$ *is the orientation given in Euler angles*[2]*, and*
- *Handedness* $\in \{"Left - handed", "Right - handed"\}$ *is the handedness.* □

Basically two kinds of reference frames can be distinguished. There is the **global reference frame** which is used to place (or move) objects in the Virtual Environment or in relation to each other. Coordinates in this reference frame do not depend on a particular object. In addition to the global reference frame, each object can also have its own **local reference frame** which is useful to express motion about the object in question.

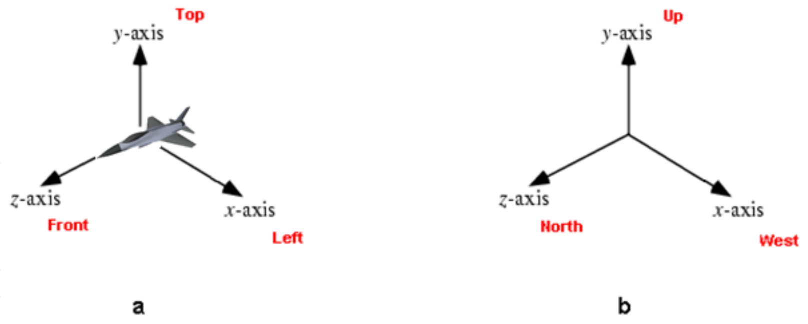


Figure 5.2: *Reference Frame*

The use of traditional axes-systems can be unintuitive for users having no background knowledge in this regard. Therefore, for laymen a natural-language-like reference frame would be better to intuitively specify positions and actions. When using a natural-language-like reference frame, a mapping needs to be made between the inherent reference frame of an object and this more intuitive reference frame based on natural-language-like directions. To achieve this, our approach allows the designer to decide for each object how to assign the directions "front", "back", "right", "left", "top", and "bottom" to the standard axes of the local reference frame. These assignments can be derived from the properties of the object itself (e.g., a car has a specific front, left, top,...). At this moment, the assignment requires human interaction. However, in the future, it could perhaps be automatically computed.

In the default scenario the x-axis is defined as the left-to-right axis of the object. The negative region represents the right of the object while the positive region represents its left. The same is done with the z-axis, describing the front and back of the object, and the y-axis, describing the top and bottom region of the object. Figure 5.2a shows this default labeling

[2]In an Euler angle representation, the axes of rotation are the axes of the local coordinate system, as opposed to the global axes.

of the axes (highlighted in red). Once this mapping has been done, the position and action of every object (relative to another object) can be expressed more intuitively in terms of these natural language directions. In this work, the assumption is made that every object has such a mapping between the standard reference frame and the natural-language-like reference frame. Furthermore, it is assumed that the origin of the local coordinate frame of an object is always defined at the centre of that object and all the axes of the local coordinate frame pass through the centre of that object (unless stated otherwise).

Apart from the local reference frame, also the global reference frame can be assigned more natural-language-like directions. In our approach, the designer can decide for the space how to assign the cardinal directions "north", "south", "east", "west", "up", and "down" to the standard axes of the global reference frame (see figure 5.2b). However, this will not be discussed in more details here since it is not used in the remaining parts of this dissertation.

The definitions in the remaining part of this section are used inside definitions in the remaining part of this chapter. The reader could skip these definitions and consult them later, when needed.

A number of definitions that will be put forward in the following sections require a unit to be given in which a particular value is expressed. Since these units are similar for different definitions, they will be defined first. We distinguish between:

- UD = {"mm", "cm", "m", "km"} is the set of distance units.
- UA = {"deg", "rot"} is the set of angle units.
- UT = {"ms", "s", "m", "h"} is the set of time units.
- UM = UD ∪ UA ∪ UT... is the set of all units of measurement.

A number of high-level modelling concepts that will be defined in this chapter, will have a number of characteristics associated to them, called *attributes*. These attributes are defined by means of a *name*, a *value* and a *unit* in which this value is expressed.

Definition 2 (Attribute). *An attribute $\mathfrak{a}$ is a triple ⟨Name, Value, Unit⟩ with:*

- *Name* $\in \mathbb{S}$ *is the name of the attribute,*
- *Value* $\in \mathbb{S} \cup \mathbb{R} \cup \mathbb{N}$ *is the value of the attribute, and*
- *Unit* $\in$ ***UM*** *is the unit in which the attribute is expressed.*

The following function symbols are defined in this context:

- *Name(a) gives the name* ***Name*** *of the attribute a*
- *Value(a) gives the value* ***Value*** *of the attribute a*
- *Unit(a) gives the unit* ***Unit*** *of the attribute a* □

As was mentioned in chapter 3, the modelling approach used by VR-WISE is an Object-Oriented (OO) approach. Therefore, the main modelling concepts are *concepts* and *instances*. Concepts are comparable with classes (or object types); instances are comparable with class instances (or objects). Concepts are used to model (describe) the relevant concepts of the application domain while instances are used to represent the actual objects in the VE. Since these concepts are used in the definitions that follow, we will define them first.

A concept is completely specified by means of its *name* and a number of *attributes* characterizing the concept. The attributes that are given for a concept are default attributes and can be overridden for individual instances of the concept.

Definition 3 (Concept). *A concept* `et` *is defined as a pair* $\langle Name, \boldsymbol{ATTRIBUTE}\rangle$ *with:*

- *Name* $\in \mathbb{S}$ *is the name of the concept, and*
- ***ATTRIBUTE*** *is the set of attributes (as defined in definition 2) of the concept where the values of the attributes represent the default values for the concept.*

The following function symbol is defined in this context:

- *Attribute(et) gives the set of attributes* ***ATTRIBUTE*** *of the concept et* □

An instance is defined by means of its *name*, its *type* and a number of *attributes*. The type refers to a concept that was defined earlier. The attributes are the actual attributes for the object.

Definition 4 (Instance). *An instance* `ei` *of a concept is defined as a triple* $\langle Name, Concept, \boldsymbol{ATTRIBUTE}\rangle$ *with:*

- *Name* $\in \mathbb{S}$ *is the name of the instance,*
- *Concept is the concept (as defined in definition 3) this instance is an instantiation of,*
- ***ATTRIBUTE*** *is a set of attributes representing the set of actual attributes of the instance, and*

- *the following must hold:* $\forall\ a_{ei} \in ATTRIBUTE, \exists!\ a_{Concept} \in Attribute(Concept)\ (Name(a_{Concept}) = Name(a_{ei}) \wedge Unit(a_{Concept}) = Unit(a_{ei}))$.

The following function symbol is defined in this context:

- *Attribute(ei) gives the set of attributes* $\boldsymbol{ATTRIBUTE}$ *of the instance ei* □

Also the concept of *shape* will be used. Basically, a shape is a visual representation for an object in the Virtual Environment. To completely specify a shape, a *name*, a *filename*, and a set of *attributes* are needed. The filename refers to a location (file) that contains the actual shape (e.g., in our library of the available shapes). The attributes are used to denote the different geometrical properties of the shape (e.g., height, width, depth,. . .) as well as the default values of these properties.

Definition 5 (Shape). *Let* $\boldsymbol{FILE} \subset \mathbb{S}$ *be the set of all filenames. A shape* `s` *is defined as a triple* $\langle Name,\ File,\ \boldsymbol{ATTRIBUTE}\rangle$ *with:*

- $Name \in \mathbb{S}$ *is the name of the shape,*

- $File \in \boldsymbol{FILE}$ *is the filename referring to a place containing the actual shape, and*

- $\boldsymbol{ATTRIBUTE}$ *is a set of attributes describing the properties of the shape.*

The following function symbol is defined in this context:

- *Attribute(s) gives the set of attributes* $\boldsymbol{ATTRIBUTE}$ *of the shape s*□

Finally, we will give a definition for the concept of *constraint*. The constraints to which we need to refer are constraints defined for complex objects. Since the topic of complex objects is outside the scope of this dissertation, an exact definition for these constraints is also outside the scope of this dissertation. More information about this topic can be found in [Bille, 2007].

Definition 6 (Constraint). *A constraint* `cc` *on a connection relation between two (or more objects) imposes a limitation to the degrees of freedom of the connected objects with respect to each other.*
The following function symbol is defined in this context:

- *Limit(cc) gives the set of named limits of the constraint cc* □

5.2 Structure Chunk

A *Structure Chunk* is a small diagram used for specifying the spatial configuration of a number of objects at a particular moment in time. In a sense, it could be considered as a smaller version of a *Static Structure Diagram* which describes the complete static scene of the Virtual Environment. Describing the complete static scene will not be detailed further on since it is not the topic of this dissertation. A Structure Chunk only describes the static scene for a small subset of the objects. Basically, a Structure Chunk is specified using a kind of graph consisting of a number of *item* concepts connected to each other by means of *relations*. Creating a Structure Chunk in the context of this dissertation typically starts with a number of items representing objects that are already in the Virtual Environment or that should become part of it at a particular time. Afterwards, relations are defined to describe how these items (objects) are related to each other from a static world point of view. This section describes the most important modelling concepts that can be used in a Structure Chunk. More formally, a Structure Chunk is defined as follows:

Definition 7 (Structure Chunk). *A Structure Chunk* $\mathtt{SC}$ *is defined as a pair* $\langle$***ITEM, RELATION***$\rangle$ *where:*

- ***ITEM*** *is a set of items, and*
- ***RELATION*** *is a set of relations on* ***ITEM****.*

The following function symbols are defined in this context:

- *Item(SC) gives the set of items* ***ITEM*** *of the SC*
- *Relation(SC) gives the set of relations* ***RELATION*** *of the SC* $\square$

In the following sub-sections, both the item and different kinds of relations are defined.

5.2.1 Item

The most important concept in a Structure Chunk is *item*. An item represents an actor that is involved in the spatial layout under definition. The concept of actor will be discussed later but for now one can think of it as an ordinary object. The concept of item is used instead of the actor itself. It can be considered as a placeholder. In fact, an item can be compared to an interface in Java (or C#). This allows the Structure Chunks to be built independently from the rest of the models. By using items instead of the actors themselves, an additional feature is that one can define a number of predefined Structure Chunks which can play the role of *layout configuration.* A layout configuration is similar to templates, describing a particular basic static structure (layout) that is used often. These layout configurations can be saved separately and instantiated as many times as needed. An example of a layout configuration is an item in a queue or on a line; or items standing in a mathematical figure such as a circle, a square and so on. This will become clearer when discussing the different relations below.

Definition 8 (Item). *An item i is simply defined as a monad $\langle Name \rangle$ where $Name \in \mathbb{S}$ is the name of the item.* □

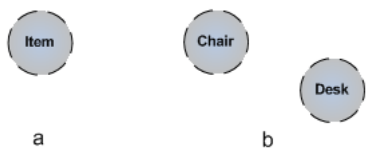


Figure 5.3: *Item (a) and example Items (b)*

An item is graphically represented by means of a dashed-line circle with the name of the item written inside the circle (see figure 5.3a).

The example in figure 5.3b defines two items called *Chair* and *Desk*. For the moment, they are unrelated. In this case, concrete names are taken but if one would like to create a more abstract layout configuration to be applicable in many applications, one could also use more abstract names that are based on the function (location) of this particular item in the Structure Chunk (e.g., LeftItem, RightItem,...).

5.2.2 Relations

Using items is not enough to specify a (part of a) static scene. The items also need to be connected to each other to express a certain relationship between them. These relations are basically meant to create either object configurations (spatial relation and orientation relation) or complex objects (connection relations). More formally:

Definition 9 (Relation). *A Relation is either a Spatial Relation, an Orientation Relation or a Connection Relation.* □

In this section, the possible relation types are discussed in more detail, and the spatial relation and orientation relation in particular.

5.2.2.1 Spatial Relations

A lot of research is being done about how people perceive the space around them. In order to act effeciently inside space, a kind of mental model of the space is created. A detailed explanation can be found in [Tversky, 2000]. These mental models of space are constructions based on elements, the things in space, and the spatial relations among them relative to a reference frame. A *spatial relation* specifies how some object is located in space with respect to some reference object. Strictly speaking, they represent relations between spatial entities. A spatial relation is an n-ary relation, a binary at least. In order to establish a spatial relation, first of all a located object (LO) is needed, also called the primary object. This is the object one wants to localize with respect to one or more other objects, different from the first. These are often called the reference objects (REFO). At least one reference object is needed but there could be more of them in special relations. In addition, the relation needs to be further qualified with a direction, usually expressed by means of spatial prepositions (relations), and a distance to locate the object [Herskovits, 1987]. The use of spatial relations which is known as a location-by-reference, provides a more intuitive means to describe the location of objects as opposed to the numerical approach (e.g., in CAD/CAM and most VR modelling packages) which is known as location-by-coordinate and where all the objects are located at an exact position (given by coordinates) in the environment. For example, using location-by-reference one can say that object A is in front of object B while using location-by-position, one has to say that object A is at position `(0,0,6)` and object B is at position `(0,0,3)`.

Definition 10 (Spatial Relation). *Let* $\boldsymbol{D}_1 = \{$*"in-front-of", "behind",* $\emptyset\}$, $\boldsymbol{D}_2 = \{$*"above", "below",* $\emptyset\}$, $\boldsymbol{D}_3 = \{$*"left-of", "right-of",* $\emptyset\}$ *be the sets of possible directions. For a Structure Chunk SC, a spatial relation* $\boldsymbol{sr}$ *is defined as a 6-tuple* $\langle Item_s,\ Item_t,\ Direction,\ Distance,\ DistanceUnit,\ RF\rangle$ *where:*

- $Item_s \in Item(SC)$ *is the source item,*
- $Item_t \in Item(SC)$ *is the target item,*
- $Direction \in \boldsymbol{D}_1 \times \boldsymbol{D}_2 \times \boldsymbol{D}_3 \setminus \{\langle\emptyset,\ \emptyset,\ \emptyset\rangle\}$ *is the direction,*
- $Distance \in \mathbb{R}$ *is the value for the distance,*
- $DistanceUnit \in \boldsymbol{UD}$ *is the unit in which the distance is expressed, and*
- *RF is the reference frame used.* □

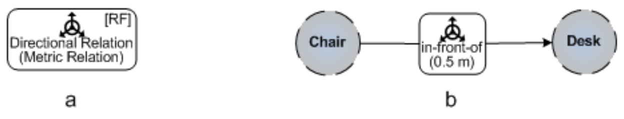


Figure 5.4: *Spatial Relation*

A spatial relation is graphically drawn as a rounded rectangle containing an icon denoting that the relation is a spatial relation (see figure 5.4a). Below the icon, there is room for the two most important attributes of a spatial relation. That is, the direction and a distance. For the direction, a number of possible values can be given: *left-of*, *right-of*, *in-front-of*, *behind*, *above*, *below*. Other concepts like the cardinal directions *north*, *east*, *south* and *west* follow the same sort of principle and could also be included. The spatial relations can be combined in order to express the position of an object more accurately. This can be done by specifying more than one direction within the same relationship. The distance is used to make the specification more precise and specifies the distance between the two objects. It is given by means of a value together with a unit (e.g., cm, m,...). An optional reference frame (RF) can be given in the upper right corner of the notation in order to specify if some other reference frame is used for the relation (such as the Terrestrial Reference Frame,...). If no reference frame is given, the local one is taken by default. The spatial relation rectangle is connected with the graphical representation of the two items involved in the spatial relation.

Figure 5.4b gives an example of a spatial relation between an item *Chair* and an item *Desk* stating that the chair is positioned *in-front-of* the desk at a distance of 0.5 m. The direction of the arrow used for the connection line indicates the order in which the relation should be read and determines which item is the source and which is the target. The source represents the to-be-located item and the target is the reference item. In figure 5.4b, the relation is read as: "The Chair is 0.5 meters in front of the Desk".

5.2.2.2 Orientation Relations

Next to a position, an object also requires an orientation to be properly placed inside the Virtual Environment. Objects thus also need to be orientated in some way. Therefore, the concept of orientation relation has been introduced. An *orientation relation* can be used to orientate an object with respect to another object by specifying (which part of) which side of the object is orientated to (which part of) which side of the other object. With the orientation relations, the VR-WISE approach wants to give the designer a more intuitive way to orientate the objects in the scene next to the traditional way of orientating objects by means of a set of exact angles around their axes.

Definition 11 (Orientation Relation). *Let* $\boldsymbol{S}_1 = \{\text{"front"}, \text{"back"}, \emptyset\}$, $\boldsymbol{S}_2 = \{\text{"left"}, \text{"right"}, \emptyset\}$, $\boldsymbol{S}_3 = \{\text{"top"}, \text{"bottom"}, \emptyset\}\}$ *be the set of possible sides. For a Structure Chunk SC, an orientation relation* or *is defined as a quadruple* $\langle Item_s, Item_t, Side_s, Side_t\rangle$ *where:*

- $Item_s \in Item(SC)$ *is the source item,*

- *$Item_t \in Item(SC)$ is the target item, and*
- *$Side_s$, $Side_t \in \boldsymbol{S}_1 \times \boldsymbol{S}_2 \times \boldsymbol{S}_3 \setminus \{\langle\emptyset, \emptyset, \emptyset\rangle\}$ indicate the side for the source item and for the target item respectively.* $\square$

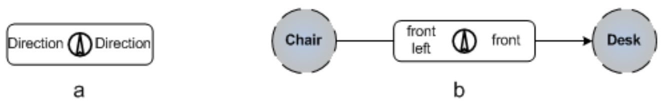


Figure 5.5: *Orientation Relation*

In the graphical notation, the orientation relation is specified using a rounded rectangle containing the orientation icon which divides the rectangle into two parts (see figure 5.5a). Each part will be connected to a particular item and inside each part, a side is specified referring to the side of the connected item to consider. Possible values that can be used for the side are *front*, *back*, *left*, *right*, *top* and *bottom*. Also here, the sides can be combined to give a more accurate specification.

Figure 5.5b expresses the orientation of the Chair and Desk from the previous example. In this example, there is an orientation relation between the Chair with the *front-left* side and the Desk with the *front* side. Also here the arrow on the connection line determines the roles of both items; the source is the to-be-orientated item and the target is the reference item. Furthermore, the arrow also indicates the reading order. In figure 5.5b the relation is read as: "The front-left side of the Chair is directed towards the front side of the Desk".

5.2.2.3 Other Relations

The spatial relations and orientation relations only allow specifying global object configurations. That is, on one hand scenes in general, and on the other hand so-called *connectionless complex objects* where complex objects are built up from different sub-objects that are not physically connected. However, those are not the only relations that are available for the specification of a Structure Chunk. As mentioned in chapter 3, the VR-WISE approach also contains a number of modelling concepts capable of describing so-called *connected complex objects*. Connected complex objects are composed of sub-objects that are physically connected. They can be specified by means of *connection relations* and *constraints* can be specified on top of these connections. The way in which the sub-objects are connected will be reflected in the behaviour of these objects, e.g., if one sub-object is moved, the other needs to follow accordingly.

A more detailed discussion of the connection relations (and the constraints on these relations) is outside the scope of this dissertation. This is the subject of another PhD thesis. The reader is referred to the work done by Bille [Bille, 2007] for more details on defining connected complex objects through a graphical notation.

5.3 Behaviour Definition Diagram

A *Behaviour Definition Diagram (BDD)* is used to describe at a conceptual level the behaviour of the objects inside the Virtual Environment. Specifically, it describes the different objects involved in a behaviour. A behaviour consists of a number of actions. The actions that the objects undertake are often a result of reacting to some external event. As briefly mentioned in the previous chapter, the behaviour is defined separated from the events that will trigger them and also separated from the specification of the actual objects that will undertake the behaviour. A BDD is graphically represented as a graph containing a set of *behaviours* (actions) interconnected through *operators* and attached to *actors*. Typically, a BDD is created by defining an actor, specifying the different properties for this actor, and then defining the different behaviours for this actor. There can be multiple behaviours defined for one actor. The behaviours itself are usually composite behaviours, which consist of a number of smaller behaviours (either simple, or composite ones) combined by means of operators. This section will discuss the modelling concepts involved in the specification of a BDD. A number of modelling concepts can be completed with small textual scripts but this will not be described here, it will be described in section 5.5. More formally, a Behaviour Definition Diagram is defined as follows:

Definition 12 (Behaviour Definition Diagram). *A Behaviour Definition Diagram* `BDD` *is defined as a quadruple* $\langle \boldsymbol{ACTOR}, \boldsymbol{BEHAVIOUR}, \boldsymbol{OPERATOR}, \boldsymbol{LINK} \rangle$ *where:*

- $\boldsymbol{ACTOR}$ *is a set of actors,*
- $\boldsymbol{BEHAVIOUR}$ *is a set of behaviours,*
- $\boldsymbol{OPERATOR}$ *is a set of operators between behaviours, and*
- $\boldsymbol{LINK}$ *is the set of links between any two actors or behaviours.*

The following function symbols are defined in this context:

- *Actor(BDD) gives the set of actors* $\boldsymbol{ACTOR}$ *of the BDD*
- *Behaviour(BDD) gives the set of behaviours* $\boldsymbol{BEHAVIOUR}$ *of the BDD*
- *Operator(BDD) gives the set of operators* $\boldsymbol{OPERATOR}$ *of the BDD*

- *Link(BDD) gives the set of links **LINK** of the BDD* □

The different components of this definition (actors, behaviours, operators, and links) will be defined in the following sections. First, we define a link.

Definition 13 (Link). *For a Behaviour Definition Diagram BDD, a link l is defined as a triple ⟨Source, Target, Sort⟩ where:*

- *Source ∈ Actor(BDD) ∪ Behaviour(BDD) is the source element,*
- *Target ∈ Actor(BDD) ∪ Behaviour(BDD) is the target element, and*
- *Sort ∈ $\mathbb{S}$ is the type of link.*

The following function symbols are defined in this context:

- *Source(l) gives the source Source of link l*
- *Target(l) gives the target Target of link l*
- *Sort(l) gives the sort Sort of link l* □

In the following sub-sections, the actor, the different behaviours (actions), operators, and special links are defined.

5.3.1 Actor

The *actor* is an important modelling concept in the specification of a Behaviour Definition Diagram. An actor represents an object that is involved in a behaviour. An actor can play different roles in a Behaviour Definition Diagram. It can act as the object for which behaviour is defined; it can act as a reference object; or it can act as an input or output object (see later). To separate the definition of a behaviour from the actual definition of the structure of an object, actors are used in the definition of a behaviour instead of the actual object(s). They are comparable to placeholders for objects or to abstract objects.

For an actor, both static and dynamic properties can be specified. The static properties are represented by a set of attributes. The dynamic properties are defined by means of the behaviours that are associated with the actor. For an actor, we only indicate the minimal set of static properties needed for specifying a behaviour. This means that later on (in the Behaviour Invocation Diagram), the actor may be replaced by objects that have at least this minimal set of properties. This can be compared with an abstract class in Java (or C#). It declares a kind of protocol on the properties meaning that each object that has those minimal properties can replace the actor and thus have the defined behaviour. This will be further explained in section 5.4.

Definition 14 (Actor). *An actor* e *is defined as a pair* $\langle Name, \boldsymbol{PROPERTY}\rangle$ *where:*

- $Name \in \mathbb{S}$ *is the name of the actor, and*
- $\boldsymbol{PROPERTY}$ *is the set of properties of the actor.*

The following function symbol is defined in this context:

- $Property(e)$ *gives the set of properties* $\boldsymbol{PROPERTY}$ *of actor* e □

Definition 15 (Property). *A property* p *is defined as a pair* $\langle Name, Unit\rangle$ *where:*

- $Name \in \mathbb{S}$ *is the name of the property, and*
- $Unit \in \boldsymbol{UM}$ *is the unit in which the property is expressed.*

The following function symbols are defined in this context:

- $Name(p)$ *gives the name Name of property* p
- $Unit(p)$ *gives the unit Unit of property* p □

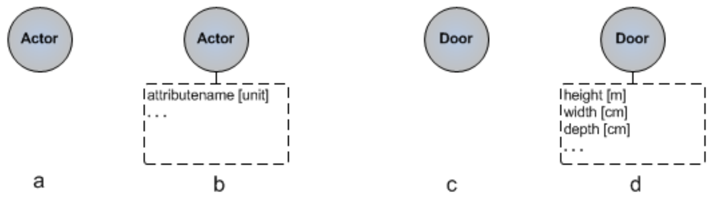


Figure 5.6: *Actor (a), detailed actor (b), example (c), detailed example (d)*

An actor is represented by means of a solid-line circle with the name of the actor written inside the circle (see figure 5.6a). When more details are needed (properties), an additional compartment is attached to the circle (see figure 5.6b). This compartment can hold the static properties of the actor. The properties are specified in a declarative manner using the following syntax: *name [unit]* where *name* is the name of an attribute and *unit* is the unit in which the value of this attribute is expressed (e.g., height[m]). Please note that our system is not explicitly typed and thus the designer is not bothered with giving a type for the attributes. This could facilitate the usage by designer not experienced in VR (or in programming).

The example in figure 5.6c defines an actor called *Door*. For this actor, a number of properties can be specified such as the *height*, *width*, *depth* and so on as shown in figure 5.6d.

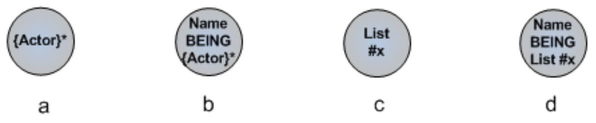


Figure 5.7: *Alternative representations of actor*

Another feature introduced in our notation is the notion of a list structure. When a large number of actors of the same kind need to be represented, a list structure can be used. Such a list of actors is represented in a similar way as a regular actor (by a circle) but the name of the actor is included in the notation: $\{\ldots\}^*$. See figure 5.7a for the abstract representation of this. In behaviour definitions, the whole list can be referenced by means of this representation. A list can also be named to be referenced later through the *name* ***BEING*** *{Actor}** expression where *name* is the name of the list (see figure 5.7b). A particular item from the list can then be represented by specifying the name of the list together with the index of the item needed. This is done using the *list #x* syntax where x is the index of the item (see figure 5.7c). Additionally, a name can be given to the item to allow easy referencing. This is done using the *name* ***BEING*** *list #x* expression where *name* is the name of the element (see figure 5.7d). The list concept helps the users in creating more powerful behaviour specifications and at the same time, it reduces the complexity of the diagrams since otherwise many similar actors need to be specified.

5.3.2 Generalization/Specialization

Actors may be classified into *generalization/specialization* hierarchies. A parent actor represents a generalization of a child actor and vice versa, a child actor is a specialization of a parent actor. When there exists an *is-a* relationship between the child actor and the parent actor, all the properties, namely the attributes as well as the behaviours defined for the parent actor are inherited by the child actor. Properties inherited from a parent actor may be overridden by the child actor. This is indicated by re-specifying the same property for the child actor.

Definition 16 (Generalization/Specialization). *For a Behaviour Definition Diagram BDD, a generalization/specialization link* $\imath$ *is a link where:*

- *Source(l)* $\in$ *Actor(BDD) is the child actor,*
- *Target(l)* $\in$ *Actor(BDD) is the parent actor, and*
- *the following must hold: Sort(l) = "subActorOf".* □

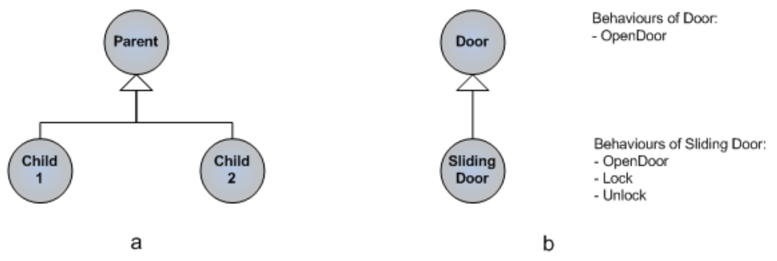


Figure 5.8: *Sub-actor/super-actor*

Generalization is represented graphically by a solid-line connector from the child (the more specific element, the sub-actor) to the parent (the more general element, the super-actor), with a large hollow triangle at the end of the connector where it meets the more general element.

See figure 5.8a for this graphical notation. Figure 5.8b gives an example showing the *Door* actor as the parent actor and the *Sliding Door* as the child actor. Suppose the *Door* actor has a behaviour called *OpenDoor*, then this behaviour is inherited by *Sliding Door*. Here the behaviour is overridden by the *Sliding Door* actor because it also appears as a property of *Sliding Door*. Furthermore, compared to *Door*, *Sliding Door* has additional behaviours called *Lock* and *Unlock*.

5.3.3 Behaviours

A behaviour can be defined for an actor. This is achieved by attaching a behaviour to an actor (or multiple actors) through a so-called *has-behaviour link* meaning that this actor will now have this behaviour as one of its dynamic properties. An actor can have multiple behaviours attached to it and vice versa, a behaviour can be attached to multiple actors through this type of link. Our approach distinguishes between *primitive behaviours* (also called *actions*) and *composite behaviours*. We first define behaviour in general; next the different types of behaviours are discussed and defined. Firstly, a number of primitive behaviours are explained. Secondly, the composite behaviours are dealt with.

Definition 17 (Behaviour). *A Behaviour is either a Move, Turn, Roll, Resize, Position, Orientate, Custom, Transform, Construct, Destruct, Ungrouping, Grouping, Disperse, Combine or a Composite Behaviour.* □

In this section, the possible behaviour types are discussed in more detail. Firstly, a number of primitive behaviours are explained. Secondly, the composite behaviours are dealt with.

5.3.3.1 Actions (Primitive Behaviours)

On the one hand, the primitive behaviours (or actions) can be categorized into manipulations and transformations that perform changes at the object level but which do not infer structural changes into the scene graph [Pellens et al., 2005b]. On the other hand, there are also behaviours such as the construct/destruct, the group/ungroup and the disperse/combine, which perform changes that have a direct influence on the structure of the overall scene graph [Pellens et al., 2006c].

Manipulations

The different actions under the manipulations category are the following:

The **move** behaviour can be used to express a change in the position of an object. To completely specify a move, a *direction* and a *distance* are needed. The *direction* specifies the (part of the) axis on which the object should move. Possible directions are: *left*, *right*, *forward*, *backward*, *up* and *down*. Multiple directions can be combined to form a more specific direction with this restriction that two directions from the same axis cannot be combined. The distance parameter expresses the distance to be travelled and should be given by means of a value and a unit (e.g., meter).

Definition 18 (move). *Let* $\boldsymbol{D}_{fb} = \{\text{"forward"}, \text{"backward"}, \emptyset\}$, $\boldsymbol{D}_{lr} = \{\text{"left"}, \text{"right"}, \emptyset\}$, $\boldsymbol{D}_{tb} = \{\text{"up"}, \text{"down"}, \emptyset\}$ *be the sets of possible directions. A move action* `ma` *is defined as a triple* $\langle$*MoveDirection, Distance, DistanceUnit*$\rangle$ *where:*

- *MoveDirection* $\in \boldsymbol{D}_{fb} \times \boldsymbol{D}_{lr} \times \boldsymbol{D}_{tb} \setminus \{\langle\emptyset, \emptyset, \emptyset\rangle\}$ *is the direction,*
- *Distance* $\in \mathbb{R}$ *is the value for the distance, and*
- *DistanceUnit* $\in$ ***UD*** *is the unit in which the distance is expressed.* □

The **turn** is used to express a rotation of the object around its top-to-bottom axis. To completely specify a turn, a *direction* and an *angle* are needed. For a turn behaviour, the value for the *direction* can only be *left* or *right*. This is because a turn of an object is only possible around a single axis, namely the top-to-bottom axis. The *angle* parameter is needed to specify how much the object needs to be turned. It should be given by means of a value and a unit.

Definition 19 (turn). *Let* $\boldsymbol{D}_{lr} = \{\text{"left"}, \text{"right"}, \emptyset\}$ *be the set of possible directions. A turn action* `ta` *is defined as a triple* $\langle$*TurnDirection, Angle, AngleUnit*$\rangle$ *where:*

- *TurnDirection* $\in \boldsymbol{D}_{lr} \setminus \{\emptyset\}$ *is the direction,*

- *Angle* $\in \mathbb{R}$ *is the value for the angle, and*
- *AngleUnit* $\in$ ***UA*** *is the unit in which the angle is expressed.* □

The **roll** is used to express the rotation of an object around either its left-to-right axis and/or its front-to-back axis. Also here, a *direction* and an *angle* are needed. In the former case (a rotation around the left-to-right axis), the value for the *direction* can be either *forward* or *backward* and in the latter case (a rotation around the front-to-back axis) this value can be either *left* or *right*. Intermediate directions can be specified by combining a direction from one axis with a direction from another axis. The *angle* parameter is again needed to denote how much the object needs to be rotated and is given by a value and a unit.

Definition 20 (roll). *Let* $\boldsymbol{D}_{fb} = \{$*"forward", "backward",* $\emptyset\}$, $\boldsymbol{D}_{lr} = \{$*"left", "right",* $\emptyset\}$ *be the sets of possible directions. A roll action* `ra` *is defined as a triple* ⟨*RollDirection, Angle, AngleUnit*⟩ *where:*

- *RollDirection* $\in \boldsymbol{D}_{fb} \times \boldsymbol{D}_{lr} \setminus \{\langle\emptyset, \emptyset\rangle\}$ *is the direction,*
- *Angle* $\in \mathbb{R}$ *is the value for the angle, and*
- *AngleUnit* $\in$ ***UA*** *is the unit in which the angle is expressed.* □

The **resize** can be used to describe a change in the scale of the object. A resize is specified by means of a *side* which denotes the side from which the object is actually being 'pulled' or 'pushed'. The value for side can be: *front*, *back*, *left*, *right*, *top* or *bottom*. Multiple sides can be specified in a single resize action. This means that the object is either pulled or pushed from all these sides simultaneously. Next, a *method* parameter is needed to denote whether it is becoming larger or smaller. The value for the method can be either *shrink* or *blowup*. Finally, an *amount* parameter is needed to denote the actual amount, in absolute figures or in percents, with which the object needs to be stretched or compressed.

Definition 21 (resize). *Let* $\boldsymbol{F} = \{$*"front", "back", "left", "right", "up", "down"*$\}$ *be the set of possible sides. A resize action* `sa` *is defined as a quadruple* ⟨***SIDE****, ResizeMethod, Amount, ResizeUnit*⟩ *where:*

- ***SIDE*** $\subseteq \boldsymbol{F}$ *and* ***SIDE*** $\neq \emptyset$ *is a nonempty set of sides,*
- *ResizeMethod* $\in \{$*"shrink", "blowup"*$\}$ *is the resizing method used,*
- *Amount* $\in \mathbb{R}$ *is the value for the amount, and*
- *ResizeUnit* $\in$ ***UD*** $\cup \{$*"%"*$\}$ *is the unit in which the amount is expressed.* □

The **position** is an action that allows the designer to describe a specific location for an object. Positioning is done by specifying the *location* by means of exact coordinates in the global reference frame of the Virtual Environment. Also the *method* saying whether the object moves suddenly or following a smooth path has to be specified. The values for method can be "at-once" or "smooth" respectively.

Definition 22 (position). *A position action* $\mathtt{pa}$ *is defined as a pair* ⟨*Location, Method*⟩ *where:*

- *Location* $\in \mathbb{R} \times \mathbb{R} \times \mathbb{R}$ *is a coordinate for the position, and*
- *Method* $\in$ {*"at-once", "smooth"*} *is the position method used.* □

The **orientate** is an alternative for the turn and the roll and allows specifying a new orientation for an object. It is specified by means of giving an *orientation* through exact angles for each of the three axes (Euler angles). There are other ways of representing an orientation but this method tends to be more intuitive for laymen. Also the *method* for the orientate action has to be specified. The value can be either "at-once" or "smooth" for respectively making the rotation at one sudden moment or gradually.

Definition 23 (orientate). *An orientate action* $\mathtt{oa}$ *is defined as a pair* ⟨*Orientation, Method*⟩ *where:*

- *Orientation* $\in \mathbb{R} \times \mathbb{R} \times \mathbb{R}$ *is an orientation in Euler angles, and*
- *Method* $\in$ {*"at-once", "smooth"*} *is the method used for performing the action.* □

Clearly, not all actions (behaviours) can be predefined. Therefore, there is possibility to define custom behaviours. The **custom** behaviour allows designers to incorporate newly defined actions when different behaviours are required than the ones provided. These behaviours are coded and thus requires programming skills. They are regarded as a normal action in our approach. The custom behaviour is not a means to compose different predefined actions together in order to create a more complex behaviour. This type of action is specified by means of a *name* identifying the action and a *script* holding the filename referring to the location where the code specifying the behaviour can be found. Currently, Lua [Ierusalimschy, 2003], a well known light-weight scripting language, is supported for writing the customized behaviours.

Definition 24 (custom). *Let* $\boldsymbol{FILE} \subset \mathbb{S}$ *be the set of all filenames. A custom action* $\mathtt{ca}$ *is defined as a pair* ⟨*Name, File*⟩ *where:*

- *Name* $\in \mathbb{S}$ *is the name of the action, and*

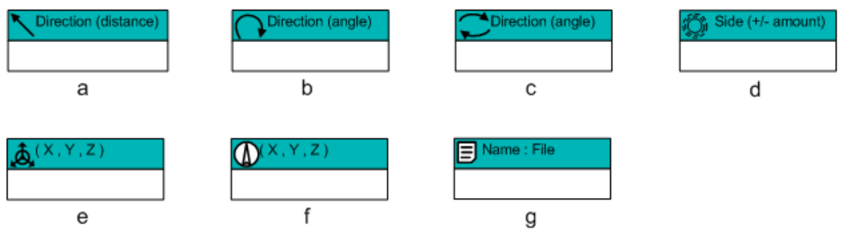


Figure 5.9: *Manipulations*

- *File* ∈ ***FILE*** *is the filename referring to the script.* □

The behaviours in this category are graphically represented by means of a rectangle with two compartments (see figure 5.9). The top compartment contains a symbol (icon) representing the kind of manipulation. The icons are created in such a way that the meaning of the actions can be intuitively derived from it. Furthermore, the ease of drawing them has also been a major consideration when designing them. The top compartment also includes the most important properties of the action itself. The second compartment is used for the textual script that can be attached to this particular action. A detailed discussion about the scripts is given in section 5.5. Figure 5.9a, 5.9b, 5.9c and 5.9d respectively show the graphical representation of the move, turn, roll and resize behaviour. Figure 5.9e and 5.9f give the graphical representation of the position and orientate behaviour. The graphical representation of the custom behaviour is shown in figure 5.9g.

Note that for this range of behaviours, but also for the following behaviours, there are actually two different modes in which they can be represented. There is the simple view in which the compartment holding the script can be hidden (or omitted) making the diagrams easier to read. And there is also the detailed view where this compartment is shown.

By default, the directions (or sides) specified for the first four manipulations, namely move, turn, roll and resize, are the directions (or sides) as perceived from the object's local reference frame. Note that, as mentioned earlier, each object has its own reference frame. However, sometimes it may be easier to specify that the object should perform the behaviour "as seen from" another object. This means that the object's local reference frame should not be used. Instead, an external reference frame should be used. In the graphical notation, attaching an actor to the behaviour by means of a so-called *reference link* indicates this. The existence of a reference link indicates that the reference frame of the reference-actor must be used to perform the primitive behaviour.

Figure 5.10a shows an example of a move behaviour with a combined direction, forward and left, over a distance of 0.5 meters. Figure 5.10b is an example of a resize behaviour which expresses an enlargement of the

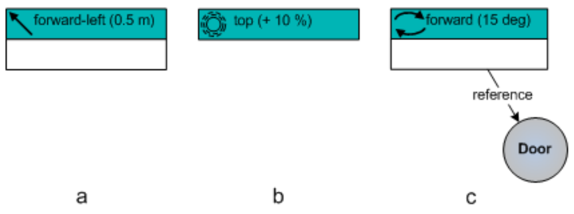


Figure 5.10: *Examples of move, resize and roll (with external reference frame)*

object from the top with an amount of 10%. Note the compartment holding the script which is omitted here. See figure 5.10c for an example of a roll behaviour towards the forward direction over an angle of 15 degrees and using a reference-actor *Door*, connected to an actor using a reference link, meaning that the manipulation is done "as seen from" the perspective of the Door. Note that in these examples the actor for which the behaviour is defined is not shown.

Transform

As seen in chapter 3, a concept (or an instance) is given a specific appearance in the Virtual Environment using the mappings. So far, this appearance was fixed throughout the complete lifetime of the application. The *transform* behaviour allows specifying how the appearance of an object can be changed (at runtime). Note that the object itself will not change, i.e. it will not suddenly become an instance of another concept but only its representation in the Virtual Environment will change. Changing the representation of an object may also imply changes in the properties of the representation. These changes can be described by means of *transformation rules.* A transformation rule represents how a property of the source object must be converted into a property of the target object. When no rules are given, a standard one-to-one transformation will be performed for the corresponding properties if possible; otherwise the defaults of the properties are taken.

Definition 25 (Transform). *Let **SHAPE** be the set of predefined shapes (as defined in definition 5). A transform action* $\boldsymbol{va}$ *is defined as a triple* $\langle Shape_s, Shape_t, \boldsymbol{RULE}\rangle$ *where:*

- $Shape_s \in \boldsymbol{SHAPE}$ *is the source shape,*
- $Shape_t \in \boldsymbol{SHAPE}$ *is the target shape, and*
- $\boldsymbol{RULE} \subseteq Attribute(Shape_t) \times Expression(Attribute(Shape_s))$ *is a set of mappings representing the transformation rules for the properties, where* $Expression(Attribute(Shape_s))$ *is a set of mathematical expressions in terms of the properties of the source shape.* □

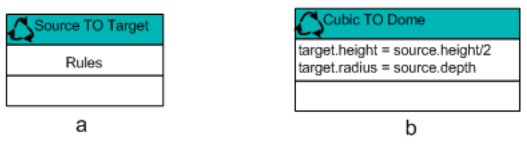


Figure 5.11: *Transform*

A transform is graphically represented as a rectangle with three compartments separated by horizontal lines (see figure 5.11a). The top compartment holds an icon representing the type of behaviour as well as the source and target representation of the object in the form of *source* ***TO*** *target*. The middle compartment holds a list of all the transformation rules needed to perform the conversion from source to target. The bottom compartment may hold a script.

The example in figure 5.11b shows the transformation of the actor from a cubic representation to a dome representation. In the conversion from one representation to the other one, two rules need to be taken into account. These are, the height of the dome is half of the height of the cube and the radius equals the depth of the cube. The properties of respectively the source and target are referred to as the *source* and *target* followed by a dot followed by the name of the property.

Construct/Destruct

An important issue in building dynamic scenes is to be able to add new objects to the Virtual Environment and to remove existing objects from the Virtual Environment, at runtime. Therefore, the primitive behaviours called the *construct* and the *destruct* are introduced.

The construct allows new objects to be added (or inserted) into the VE at runtime. Upon execution of this action, the scene graph is extended with a branch containing the new object(s). Of course, newly added objects need to be positioned and orientated correctly. A so-called *Structure Chunk* is used for this. It allows relating these new objects to already existing objects in the scene at the time of creation. Note that (at runtime) once the construct behaviour is completed the specification given in the Structure Chunk might not be valid anymore due to other changes of the scene. It is also possible to specify how the object(s) to be created, should appear in the Virtual Environment. This is done by means of the optional *method* parameter. The value for this parameter can either be *appear*, *fade-in*, *grow* or *zoom-in*. When appear is used, the object will just appear at once. Fade-in allows the object to gradually become visible. Grow and zoom-in make the object entering the Virtual Environment by gradually expanding from nothing to the actual size from either the ground (grow) or the centre (zoom-in) of the object.

Definition 26 (Construct). *For a Behaviour Definition Diagram BDD, a construct action* **ba** *is defined as a triple ⟨Name, SC, ConstructMethod⟩ where:*

- *Name ∈ $\mathbb{S}$ is the name of the action,*
- *SC is a Structure Chunk describing the position of the constructed actor, and*
- *ConstructMethod ∈ {"appear", "fade-in", "grow", "zoom-in"} is the construction method used.*

Furthermore,

- *there is a nonempty set of links $\boldsymbol{LINKS}_{ba}$ ⊆ Link(BDD) such that: ∀ l ∈ $\boldsymbol{LINK}_{ba}$ (Source(l) = ba ∧ Target(l) ∈ Actor(BDD) ∧ Sort(l) = "output"); which indicates that the construct action is attached to at least one actor through an output link.* □

Using the destruct, objects can be deleted from the VE at runtime. Note that destroying an object will not only make the object disappear from the environment, but it will also delete it from the scene graph. When the object that needs to be destroyed is part of a connectionless complex object, the relationships in which this object was involved will be deleted too; when it is part of a connected complex object, the connections in which the object is involved will be deleted as well. Similarly as for the creation of objects, when specifying this behaviour, an optional *method* parameter can be specified. The possible values for this parameter are: *disappear*, *fade-out*, *shrink* and *zoom-out*. When disappear is used, the object just disappears at once. When fade-out is taken as value, the object will gradually become invisible. Shrink and zoom-out allows removing the object by gradually becoming smaller, either towards the ground (shrink) or towards the centre (zoom-out) of the object.

Definition 27 (Destruct). *For a Behaviour Definition Diagram BDD, a destruct action* **ea** *is defined as a pair ⟨Name, DestructMethod⟩ where:*

- *Name ∈ $\mathbb{S}$ is the name of the action, and*
- *DestructMethod ∈ {"disappear", "fade-out", "shrink", "zoom-out"} is the destruction method used.*

Furthermore,

- *there is a nonempty set of links $\boldsymbol{LINKS}_{ea}$ ⊆ Link(BDD) such that: ∀ l ∈ $\boldsymbol{LINK}_{ea}$ (Target(l) = ea ∧ Source(l) ∈ Actor(BDD) ∧ Sort(l) = "input"); which indicates that the destruct action is attached to at least one actor through an input link.* □

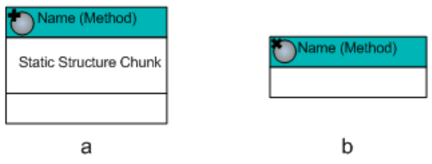


Figure 5.12: *Construct (a) and destruct (b)*

The construct behaviour is graphically represented by means of a rectangle with three areas (see figure 5.12a). The top area contains the icon representing the construct behaviour as well as the name of the behaviour and the optional method parameter. The middle area is reserved for the Structure Chunk specifying how the object must be integrated into the existing Virtual Environment, while the third area may hold a script. A construct has at least one actor attached to it, through an *output link*, representing the object that is created. The opposite behaviour, the destruct, is graphically represented like any of the previous actions, having two compartments (see figure 5.12b). The top compartment contains the icon representing the kind of action (here the destruct action) together with the name of the behaviour and optionally the method used for the destruction; the second compartment may hold a script. The destruct has at least one actor attached to it, through an *input link*, representing the object that is to be deleted from the scene.

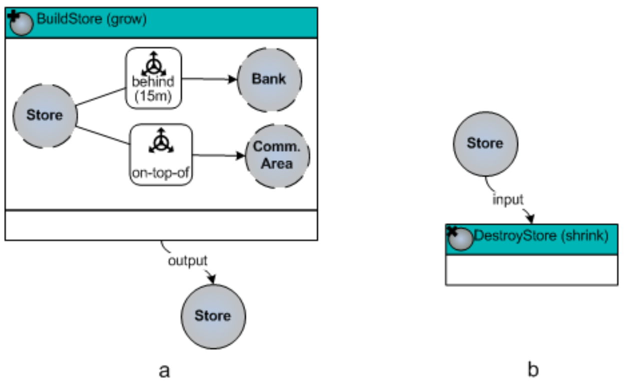


Figure 5.13: *Example of construct (a), destruct (b)*

Figure 5.13 presents an example of the construct and destruct behaviour. In the construct behaviour, called BuildStore, a new building, represented by the *Store* actor, will be added to the scene. The Structure Chunk specifies where the new building should be positioned. It will be located on the right of the Bank building at a distance of 15 meters, and on-top-of the *Commercial Area*. In this case, no orientation relation is given meaning that the Store will be given a default orientation. With the destruct behaviour, called DestroyStore, a building, represented by the *Store* actor, will be deleted from the scene.

Grouping/Ungrouping

The second kind of scene graph modification is to be able to create at runtime a group of objects or decompose at runtime an existing group of objects or alternatively, to assemble and disassemble objects in the Virtual Environment. Therefore, the primitive behaviours called *grouping* and *ungrouping* are introduced.

The grouping behaviour is used to specify that a new object (or object group) should be created by combining or assembling a number of objects at runtime. In other words, new connected complex or unconnected complex objects can be created at runtime. The objects will be taken from their current position in the scene graph and brought together under the same branch according to the given structure. An optional Structure Chunk is used to describe the structure of the new object. By using spatial relations and/or orientation relations, one can create unconnected complex objects, and by specifying connection relations[3], one can build connected complex objects at runtime. Note that in this case, in contrast with the previous construct behaviour, the Structure Chunk is used to express a complex object and therefore, the relations that will be introduced at runtime will be fixed relations. That is, after the behaviour has been performed for a number of objects, the newly created object will behave as a complex object and therefore, if one of its parts (or the object itself) is manipulated, the relations will ensure that the other parts will undergo the same manipulation as well. It is also possible to specify how the object(s) should be grouped in the Virtual Environment. This is done by means of the optional *method* property. The value for this property can either be *at-once* or *smooth*. When at-once is used, the objects will just be grouped at a sudden moment. Smooth allows the objects to fluently move towards each other.

Definition 28 (Grouping). *For a Behaviour Definition Diagram BDD, a grouping action* $\mathtt{ga}$ *is defined as a triple* $\langle$*Name, SC, Method*$\rangle$ *where:*

- *Name* $\in \mathbb{S}$ *is the name of the action,*
- *SC is a Structure Chunk describing the position of the created group, and*
- *Method* $\in$ {*"at-once", "smooth"*} *is the grouping method used.*

Furthermore,

[3]The connection relations were not discussed in this dissertation.

- *there is a nonempty set of links* $\boldsymbol{LINK}_{ga} \subseteq$ *Link(BDD) such that:* $\forall\, l \in \boldsymbol{LINK}_{ga}$ *(Target(l) = ga* $\wedge$ *Source(l)* $\in$ *Actor(BDD)* $\wedge$ *Sort(l) = "input"); which indicates that the grouping action is attached to at least one actor through an input link, and*

- *there is exactly one link* $\boldsymbol{l}_{ga} \in$ *Link(BDD) such that: Source(*l_{ga}*) = ga* $\wedge$ *Target(*l_{ga}*)* $\in$ *Actor(BDD)* $\wedge$ *Sort(*l_{ga}*) = "output"; which indicates that the grouping action is attached to exactly one actor through an output link.* □

The ungrouping behaviour is the opposite of the grouping behaviour and removes all the fixed relationships and connections between the objects used when the group was created. An optional Structure Chunk can be given to describe the new positions of the objects from the group after it has been ungrouped. Note that the relationships specified here are only valid at the time of ungrouping and might not be valid anymore once the behaviour has been completed. When no Structure Chunk is given, the objects will just remain at the same positions as they were before. Also here, an optional *method* parameter can be set to denote the way in which the objects are repositioned. The value for this parameter can either be *at-once* or *smooth*. When at-once is used, the objects will just be repositioned from one frame to the next. Smooth allows the objects to fluently float away to the proper positions.

Definition 29 (Ungrouping). *For a Behaviour Definition Diagram BDD, an ungrouping action* $\boldsymbol{ua}$ *is defined as a triple* $\langle$*Name, SC, Method*$\rangle$ *where:*

- *Name* $\in \mathbb{S}$ *is the name of the action,*

- *SC is a Structure Chunk describing the position of the separate objects, and*

- *Method* $\in$ {*"at-once", "smooth"*} *is the ungrouping method used.*

Furthermore,

- *there is exactly one link* $\boldsymbol{l}_{ua} \in$ *Link(BDD) such that: Target(*l_{ua}*) = ua* $\wedge$ *Source(*l_{ua}*)* $\in$ *Actor(BDD)* $\wedge$ *Sort(*l_{ua}*) = "input"; which indicates that the ungrouping action is attached to exactly one actor through an input link.* □

In the graphical notation, the grouping action is represented by a rectangle divided into three compartments (see figure 5.14a). In the top compartment, the icon for the grouping action is shown as well as the name of the behaviour and the optional method parameter. The middle compartment holds the Structure Chunk. The bottom compartment may hold a script. A grouping behaviour is related to one or more input actors (the items that

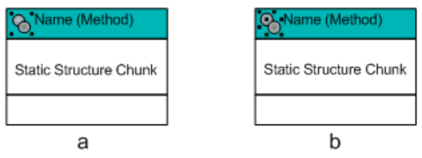


Figure 5.14: *Grouping (a) and ungrouping (b)*

should be used to form the group) and one output actor (the group itself). The ungrouping behaviour is also represented as a rectangle with three compartments (see figure 5.14b). In the first compartment, the ungroup icon is given as well as the name of the behaviour together with the optional method parameter. The middle compartment holds the Structure Chunk. The third compartment is reserved for the script. An ungrouping behaviour is linked to only one input actor, namely the actor representing the group. No output actors are necessary since the object(s) do already exist.

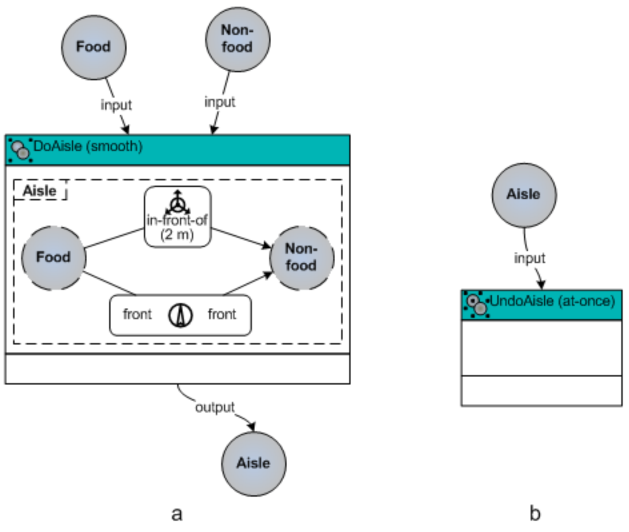


Figure 5.15: *Example of grouping (a), ungrouping (b)*

The example in figure 5.15a defines a behaviour, called DoAisle, which will place two racks, *Food* and *Non-food* in our store together to form a group called *Aisle*. The Structure Chunk specifies how the objects in the group should be positioned. The Food object will be located in front of the Non-Food object at a distance of 2 meters and both objects will be facing each other. Figure 5.15b shows an example of an ungroup behaviour, called UndoAisle, for an Aisle group. Since no explicit Structure Chunk is given here, the objects will just be ungrouped without repositioning them.

Disperse/Combine

The third type of modification in the scene graph is the ability to break at runtime an object apart into multiple smaller objects or to merge at runtime a number of objects into one single object. To do the former, the *disperse* behaviour is introduced. To do the latter, the *combine* behaviour is introduced.

The disperse behaviour allows specifying that an object should be divided into two (or more) pieces at runtime. What it actually does is that the main branch of the object in the scene graph is being deleted from the scene and new smaller branches are created for the different parts that are introduced. Like in the construct behaviour, the new objects resulting from this behaviour need to be placed correctly in the Virtual Environment. A Structure Chunk is used to specify how the newly created objects should be related to each other and how they should be related to existing objects. If a disperse behaviour is invoked on a complex object (connected or unconnected), it will remove all the relations that exist between the (parts of the) complex object. This implies that if the user manipulates one of the objects of the original complex object, the other objects of the former object will not be manipulated accordingly since there is no physical connection anymore.

Definition 30 (Disperse). *For a Behaviour Definition Diagram BDD, a disperse action* ***da*** *is defined as a pair ⟨Name, SC⟩ where:*

- *Name* $\in \mathbb{S}$ *is the name of the action, and*
- *SC is a Structure Chunk describing the dispersed object.*

Furthermore,

- *there is a nonempty set of links* $\boldsymbol{LINK}_{da} \subseteq$ *Link(BDD) such that:* $\forall\, l \in \boldsymbol{LINK}_{da}$ *(Source(l) = da* $\wedge$ *Target(l)* $\in$ *Actor(BDD)* $\wedge$ *Sort(l) = "output"); which indicates that the disperse action is attached to at least one actor through an output link, and*
- *there is exactly one link* $\boldsymbol{l}_{da} \in$ *Link(BDD) such that: Target(*l_{da}*) = da* $\wedge$ *Source(*l_{da}*)* $\in$ *Actor(BDD)* $\wedge$ *Sort(*l_{da}*) = "input"; which indicates that the disperse action is attached to exactly one actor through an input link.* □

The combine behaviour is the reverse of the disperse behaviour and allows bringing together a number of objects and merge them into one single object according to the specification that is given by means of a Structure Chunk.

Definition 31 (Combine). *For a Behaviour Definition Diagram BDD, a combine action* ***aa*** *is defined as a pair ⟨Name, SC⟩ where:*

- *Name* $\in \mathbb{S}$ *is the name of the action, and*

- *SC is a Structure Chunk describing the combined object.*

Furthermore,

- *there is a nonempty set of links* $\boldsymbol{LINK}_{aa} \subseteq$ *Link(BDD) such that:* $\forall\ l \in \boldsymbol{LINK}_{aa}$ *(Target(l) = aa* $\wedge$ *Source(l)* $\in$ *Actor(BDD)* $\wedge$ *Sort(l) = "input"); which indicates that the combine action is attached to at least one actor through an input link, and*

- *there is exactly one link* $\boldsymbol{l}_{aa} \in$ *Link(BDD) such that: Source(*l_{aa}*) = aa* $\wedge$ *Target(*l_{aa}*)* $\in$ *Actor(BDD)* $\wedge$ *Sort(*l_{aa}*) = "output"; which indicates that the combine action is attached to exactly one actor through an output link.* □

The difference between the disperse behaviour and the ungrouping behaviour is that in the latter the output objects are not created but already existed while in the former they did not exist beforehand. Also the original object will be destroyed in case of a disperse. In the same way the combine behaviour is similar to the grouping but with this difference that the input objects do not exist anymore once the behaviour has been performed.

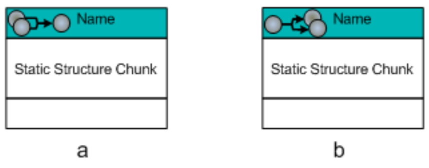


Figure 5.16: *Combine (a) and disperse (b)*

Both the disperse and the combine behaviours are graphically represented by means of a rectangle divided into three compartments (see figure 5.16a and b). The top compartment contains the icon for the action (disperse or combine) as well as the name given to this behaviour. The middle compartment is used to specify the Structure Chunk, describing the static structure of the part-objects (for the disperse) or the new object (for the combine). The third compartment may hold the script for this behaviour. The disperse behaviour has a single actor attached to it through an *input link*, representing the object that will be divided, and one or more actors through an *output link*, representing the different parts. The combine behaviour has one or more input actors, representing the parts, and only one output actor, representing the newly created object.

Figure 5.17 shows an example of a disperse and a combine action. In the first example, BreakShelve, the *Shelve* object will be partly broken on the right side. In particular, the object is dispersed into two separate objects, namely the *Shelve Piece* and the *Right Support*. According to the specification given in the Structure Chunk, the ShelvePiece will be positioned 1 cm

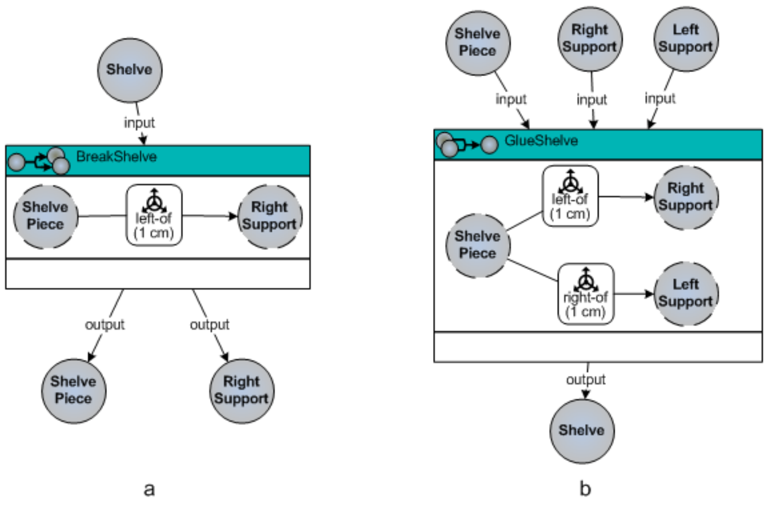


Figure 5.17: *Example of disperse (a), combine (b)*

left-of the RightSupport. The second example is the opposite of the first and glues three parts namely *Shelve Piece*, *Left Support* and *Right Support* together into one whole *Shelve* by putting the ShelvePiece 1 cm left-of the RightSupport and 1 cm right-of the LeftSupport.

5.3.3.2 Composite Behaviours

A composite behaviour is a composition of two or more behaviours (called sub-behaviours) that are combined with each other in a well-defined way. Composite behaviours enable the designer to describe more complex behaviours. A sub-behaviour of a composite behaviour can be either a primitive behaviour of any type (the actions discussed above) or a composite behaviour on its own. The composite behaviour is a very powerful modelling element. It not only allows specifying more complex behaviours, it provides the designer an effecient abstraction mechanism. Large behaviours can be broken down into smaller and less complex behaviours. Composite behaviours can be parameterized by means of specifying attributes for it. The attributes can be passed to the sub-behaviours by their name. This kind of hierarchical structuring may improve the maintainability, reusability and understandability of the behaviour specifications.

Definition 32 (Composite Behaviour). *For a Behaviour Definition Diagram BDD, a composite behaviour* `cb` *is defined as a triple* ⟨*Name*, ***SUBBEHAVIOUR***, ***ATTRIBUTE***⟩ *with:*

- *Name* $\in \mathbb{S}$ *is the name of the composite behaviour,*
- ***SUBBEHAVIOUR*** $\subset$ *Behaviour(BDD) is the set of sub-behaviours, and*

- ***ATTRIBUTE*** *is the set of attributes (as defined in definition 2) of the composite behaviour where the values of the attributes represent the default values for the composite behaviour.*

The following function symbols are defined in this context:

- *Attribute(cb) gives the set of attributes* ***ATTRIBUTE*** *of the composite behaviour cb*

- *SubBehaviour(cb) gives the set of sub-behaviours* ***SUBBEHAVIOUR*** *of the composite behaviour cb* □

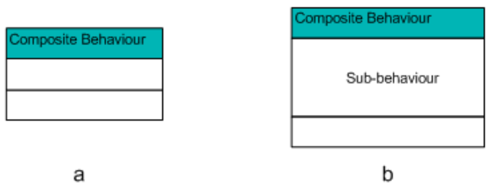


Figure 5.18: *External Behaviour (a) and Composite Behaviour (b)*

The graphical notation of a composite behaviour is a rectangle divided into three compartments (see figure 5.18b). The top compartment shows the name of the behaviour. The second compartment holds a (nested) behaviour definition diagram representing the composite behaviour (using the operators described in the next section). In some situations, it may be more convenient to hide the nested diagram of a composite behaviour. For example if it is too large. Then, the area of the nested diagram is left empty (see figure 5.18a) and the content is shown in a separate diagram elsewhere. However, using this alternative notation does not change the semantics of the composite behaviour. The third compartment holds the script and/or the attributes belonging to this composite behaviour.

Figure 5.19 shows an example of a composite behaviour. The behaviour called *SmoothLeftTurn* is composed of two sub-behaviours (actions), namely a *move* and a *turn*. At this point, it is not important to know how these two actions are combined to each other. We will give more details about this in the following section. For now, let us assume that the two actions are executed simultaneously. In the simplest case, all the information for the sub-behaviours is explicitly specified such as in the turn action. However, one can also define an attribute and parameterize the sub-behaviour with this attribute. This is the case in the move action where the attribute *distance* is defined and used in the action to replace an actual distance.

5.3.4 Operators

In the previous sections, only the basic actions and behaviours were discussed. However, in order to build complex behaviours, a mechanism is re-

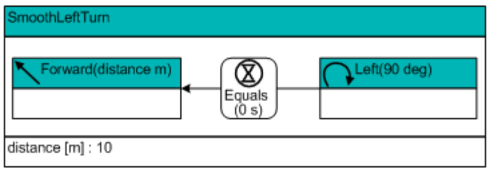


Figure 5.19: *Example of composite behaviour*

quired to combine the different primitive actions, or composite behaviours, together. In our approach, this is achieved using the concept of operator. In many approaches we discussed in the related work chapter, we found that in most cases, the mechanism for combining different behaviours was often very primitive. Therefore, we searched for different ways of combining behaviours in order to provide the designer with a more rich set of modelling concepts. Four different kinds of operators are introduced in this dissertation, the temporal operators, the lifetime operators, the conditional operator and the influential operator. We first define the concept of operator in general; next the different types of operators are explained and defined.

Definition 33 (Operator). *An Operator is either a Temporal Operator, a Lifetime Operator, a Conditional Operator, or an Influential Operator.* □

The different kinds of operators are discussed in more detail in the following sub-sections.

5.3.4.1 Temporal Operators

In the same way that people have a mental model of how objects in the Virtual Environment are related to each other, they also have a mental model of when actions should occur inside the Virtual Environment and how these actions are related to each other. The *temporal operators* are used for expressing this. A temporal operator allows specifying, in a natural way, the order of the different actions and behaviours throughout time. They provide a more intuitive way for specifying time-dependencies between behaviours than the typical keyframe animation methods found in most Virtual Reality modelling tools. Furthermore, they allow synchronizing behaviours. Different temporal models exist to describe timing. There are the point-based models in which relations only use zero-length moments in time are described [Hirzalla et al., 1995]. The interval-based models define complete time intervals for which also a number of relations are created [Allen, 1991]. A different kind of conceptual model capturing the essential semantics of time-varying information is described in [Spaccapietra et al., 1998]. The temporal operators employed here, are based on the binary interval relations. However, some adaptations were needed in order to completely specify relationships between behaviours. This is because some of the original relationships were underspecified for our purpose. Other operators were not included at this moment but could be included in the future. The operators currently supported in our approach are:

- **before(x, y, t)**: Behaviour x ends t seconds before the behaviour y starts; there is a gap of t seconds.
- **meets(x, y)**: Behaviour y starts immediately after the end of behaviour x.
- **overlaps(x, y, t)**: Behaviour y starts t seconds before the end of behaviour x; there is an overlap of t seconds.
- **during(x, y, t1, t2)**: Behaviour x starts t1 seconds after the start of behaviour y and ends t2 seconds before the end of behaviour y.
- **starts(x, y, [. . .])**: Behaviour x and behaviour y are initiated at the same moment.
- **ends(x, y, [. . .])**: Behaviour x and behaviour y finish at the same moment.
- **equals(x, y, [. . .])**: Behaviour x and behaviour y both start and finish at the same moment.

All these temporal operators, except equals, have an inverse, namely **after**, **met-by**, **overlapped-by**, **contains**, **started-by** and **ended-by**. In their standard form, the operators are all binary relations connecting two behaviours. However, some of the operators (namely starts, ends and equals) additionally have an n-ary variant meaning that one operator can connect multiple behaviours. This is indicated by the [. . .] notation.

Definition 34 (Temporal Operator). *For a Behaviour Definition Diagram BDD, a temporal operator* `to` *is defined as either:*

- *a triple* $\langle Behaviour_s, Behaviour_t, IR\rangle$ *where* $IR \in \{$*"meets", "met-by", "equals", "started-by", "ended-by", "starts", "ends"*$\}$,
- *a quintuple* $\langle Behaviour_s, Behaviour_t, IR, Time, TimeUnit\rangle$ *where* $IR \in \{$*"before", "after", "overlaps", "overlapped-by"*$\}$,
- *a 6-tuple* $\langle Behaviour_s, Behaviour_t, IR, Time_1, Time_2, TimeUnit\rangle$ *where* $IR \in \{$*"during", "contains"*$\}$,

and where:

- $Behaviour_s \in Behaviour(BDD)$ *is the source behaviour,*
- $Behaviour_t \in Behaviour(BDD)$ *is the target behaviour,*

- *Time, $Time_1$, $Time_2 \in \mathbb{R}$ are the time identifiers, and*
- *TimeUnit $\in$ **UT** is the unit in which the time identifiers are expressed.*

The following function symbols are defined in this context:

- *Source(to) gives the source behaviour $Behaviour_s$ of operator to*
- *Target(to) gives the target behaviour $Behaviour_t$ of operator to* □

5.3.4.2 Lifetime Operators

Sometimes, there are situations that it should not be possible to execute some behaviours of an object, or that it must be possible to put an executing behaviour on hold and resume it afterwards. The second type of operator is therefore the *lifetime operator*. They are meant to allow this kind of authority over behaviours. The lifetime operators allow the designer to describe that one behaviour (source) controls the lifetime (being enabled or not, active or not) of other (target) behaviour(s). The following list gives the overview of the different lifetime behaviours supported in our approach. The notation [...] is used to represent that more than two behaviours can be specified:

- **enable(x, y, [...])**: Behaviour y gets enabled when behaviour x terminates.
- **disable(x, y, [...])**: Behaviour y is disabled when behaviour x terminates; behaviour y cannot be triggered anymore until it has been enabled again.
- **suspend(x, y, [...])**: When behaviour x terminates, behaviour y is deactivated and holds its state in order to be resumed later on; behaviour y can only be resumed or disabled afterwards.
- **resume(x, y, [...])**: Behaviour y is reactivated when behaviour x terminates; it allows a behaviour to continue to operate after it has been suspended; the behaviour continues from where it has stopped.

Definition 35 (Lifetime Operator). *For a Behaviour Definition Diagram BDD, a lifetime operator* `lo` *is defined as a triple $\langle Behaviour_s$, $Behaviour_t$, Tense$\rangle$ where:*

- *$Behaviour_s \in$ Behaviour(BDD) is the source behaviour,*
- *$Behaviour_t \in$ Behaviour(BDD) is the target behaviour, and*
- *Tense $\in$ {"enable", "disable", "suspend", "resume"} is the particular type of operator used.*

The following function symbols are defined in this context:

- *Source(lo) gives the source behaviour $Behaviour_s$ of operator lo*
- *Target(lo) gives the target behaviour $Behaviour_t$ of operator lo* $\square$

5.3.4.3 Conditional Operator

There are situations where it is important to have control over the flow of the actions by means of conditions. For this, the *conditional operator* is used. When using a conditional operator, a conditional expression and all possible behaviours have to be defined. Each behaviour needs to be linked with a possible outcome (value) of the conditional expression. The behaviour that will be invoked depends on the value of the conditional expression. A condition results in *true* or *false* and this value will actually trigger the corresponding behaviour. The condition itself can be described by means of a number of relational operators ($<$, $>$, $\leq$, $\geq$, $=$). In addition, the standard arithmetic operators may be used ($+$, $-$, $/$, $*$). The conditions can also be combined or negated using the boolean operators (AND, OR, NOT). The static properties of the object involved in the behaviour can also be used. They are referenced by means of the name of the actor followed by the name of the property.

Definition 36 (Conditional Operator). *Let* $\boldsymbol{BC}$ *be the set of boolean conditions. For a Behaviour Definition Diagram BDD, a conditional operator* `co` *is defined as a quadruple* $\langle Behaviour_s, Behaviour_{t1}, Behaviour_{t2}, Condition\rangle$ *where:*

- $Behaviour_s \in$ *Behaviour(BDD) is the source behaviour,*
- $Behaviour_{t1} \in$ *Behaviour(BDD) and* $Behaviour_{t2} \in$ *Behaviour(BDD) are the target behaviours associated with the true and false condition respectively, and*
- $Condition \in \boldsymbol{BC}$ *is the boolean condition for this operator.*

The following function symbols are defined in this context:

- *Source(co) gives the source behaviour $Behaviour_s$ of operator co*
- *TargetT(co) gives the target behaviour $Behaviour_{t1}$ of operator co*
- *TargetF(co) gives the target behaviour $Behaviour_{t2}$ of operator co* $\square$

5.3.4.4 Influential Operator

In addition to the operators in which a behaviour depends on another behaviour by means of time, or by means of a condition, there might be the case in which the execution of one behaviour is influenced by the execution of another behaviour. Therefore, an *influential operator* is introduced. The influential operator indicates the inter-relationships of the behaviours of the objects involved in the relation. It also tells us how the manipulation of one object can influence the movement of another object. When using an influential operator, an expression needs to be given describing the coupling between the behaviours. The expression is described by means of a mathematical equation describing the relationship in the properties of one behaviour with the properties of another behaviour. The use of the influential operator is extremely useful when modelling mechanical devices e.g., gears, belts and pulleys. For example, in a rack-and-pinion gear (which converts a rotation into a linear motion), the pinion is rotating and this rotation engages the movement of the rack respectively according to the ratio given in the operator (e.g., x = 3/4 y).

Definition 37 (Influential Operator). *Let **ME** be the set of mathematical equations. For a Behaviour Definition Diagram BDD, an influential operator* `io` *is defined as a triple* $\langle Behaviour_s, Behaviour_t, Influence\rangle$ *where:*

- $Behaviour_s \in Behaviour(BDD)$ *is the source behaviour,*
- $Behaviour_t \in Behaviour(BDD)$ *is the target behaviour, and*
- $Influence \in \boldsymbol{ME}$ *is the mathematical equation representing the inter-relationship.*

The following function symbols are defined in this context:

- *Source(io) gives the source behaviour* $Behaviour_s$ *of operator io.*
- *Target(io) gives the target behaviour* $Behaviour_t$ *of operator io.* □

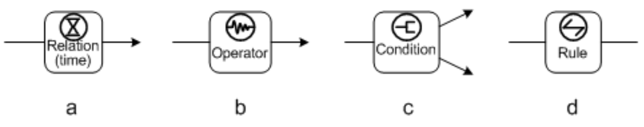


Figure 5.20: *Operators*

In the graphical notation, an operator is drawn as a rectangle with rounded corners containing an icon that indicates the type of operator (see figure 5.20). Figure 5.20a shows the notation for a temporal operator, figure

5.20b shows the notation for a lifetime operator, figure 5.20c shows the notation for a conditional operator, and figure 5.20d shows the notation for an influential operator. The graphical element is connected to the behaviours involved by single solid lines. The arrow on the connection line indicates the source and the target behaviour as well as the reading direction (see below for an example). Below the icon indicating the type of operator, there is some space available to give the parameters of the operator.

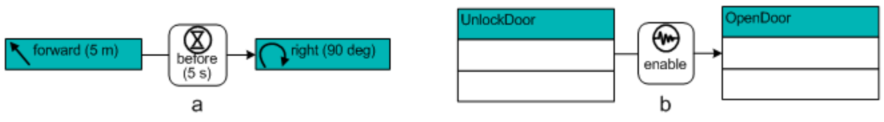


Figure 5.21: *Example of temporal operator (a) and lifetime operator (b)*

Figure 5.21a shows an example of the use of a temporal operator. The figure can be read as follows: a moving action being a move of 5 meters in the forward direction needs to happen 5 seconds before a turning action of 90 degrees to the right is performed. Figure 5.21b illustrates a lifetime operator. Here, the execution of the UnlockDoor behaviour will enable the OpenDoor behaviour (which has previously been disabled in some other way).

5.3.5 Example

In this section, an example is given which illustrates the use of a Behaviour Definition Diagram and the modelling concepts introduced in the previous sections. The example is taken from an industrial context and presents the definition of a virtual robot in a manufacturing plant (see figure 5.3.5).

There are three composite behaviours defined. For the actor, called Master, the composite behaviour ProductionCycle is defined. This behaviour has a number of sub-behaviours. It is composed of three primitive behaviours (or actions) and two composite behaviours. These sub-behaviours are combined with each other by means of the temporal operators. It performs the following scenario of synchronized movements. The object first moves 1.5 meters forward, and then immediately the composite behaviours on its parts (LeftArmMovement and RightArmMovement) are started, as well as an left movement of 0.5 meters. After the forward movement has ended, the Master turns right over 90 degrees and this movement needs to end at the same time as the left movement. The details about the composite behaviours, LeftArmMovement and RightArmMovement, are omitted in the diagram. Furthermore, the diagram also shows a powerOn and a PowerOff behaviour for a Control actor. Both these behaviours only consist of one sub-behaviour (action), they will roll over the Control over 30 degrees. Executing the PowerOn and PowerOff behaviour will enable, respectively disable, the

ProductionCycle behaviour of the Master. These last actions are specified by means of the lifetime operators enable and disable.

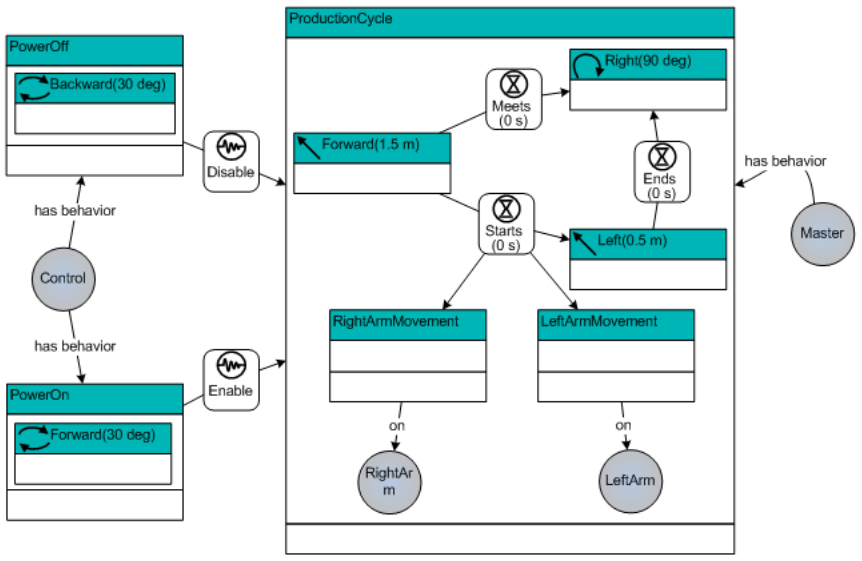


Figure 5.22: *Behaviour Definition Diagram: Virtual Robot*

5.4 Behaviour Invocation Diagram

As explained before, the definition of behaviours is independent from the specification of the actual objects in the Virtual Environment. The *Behaviour Invocation Diagram (BID)* can be seen as a sort of instantiation, at a conceptual level, of a Behaviour Definition Diagram. It parameterizes and assigns behaviours defined earlier to actual objects populating the Virtual Environment. Furthermore, it also denotes the events that may trigger these behaviours for the particular objects. This improves reusability and enhances flexibility since the same behaviour definition can be reused for different objects (if they have the same behaviour) and the same behaviour can be triggered in different ways (e.g., by some user interaction or by a collision with another object). A BID is graphically represented as a graph containing a number of *objects* (either concepts or instances) connected to *behaviour references* which are linked to *events* that may trigger them. A BID is typically created by first defining an object, and link this object to an actor defined in a BDD. As soon as this is done, the behaviours for this object are known and they can be instantiated through the behaviour references. Finally, the events are specified defining what will trigger the behaviours (behaviour references). The different components used in a BID will now be discussed. More formally, we define a Behaviour Invocation

Diagram as follows:

Definition 38 (Behaviour Invocation Diagram). *A Behaviour Invocation Diagram* `BID` *is defined as a quadruple* $\langle$***OBJECT***, ***REFERENCE***, ***EVENT***, ***LINK***$\rangle$ *where:*

- ***OBJECT*** *is a set of objects,*
- ***REFERENCE*** *is a set of behaviour references,*
- ***EVENT*** *is a set of events, and*
- ***LINK*** *is a set of links between any two objects, behaviour references or events.*

The following function symbols are defined in this context:

- *Object(BID) gives the set of objects* ***OBJECT*** *of the BID*
- *Reference(BID) gives the set of behaviour references* ***REFERENCE*** *of the BID*
- *Event(BID) gives the set of events* ***EVENT*** *of the BID*
- *Link(BID) gives the set of links* ***LINK*** *of the BID* $\square$

In the following sub-sections, the different elements, namely object, behaviour reference, events, and special links are further defined. First, we define the concept link in the context of a Behaviour Invocation Diagram.

Definition 39 (Link). *For a Behaviour Invocation Diagram BID, a link* $\imath$ *is defined as a triple* $\langle$*Source, Target, Sort*$\rangle$ *where:*

- *Source* $\in$ *Object(BID)* $\cup$ *Reference(BID)* $\cup$ *Event(BID) is the source element,*
- *Target* $\in$ *Object(BID)* $\cup$ *Reference(BID)* $\cup$ *Event(BID) is the target element, and*
- *Sort* $\in \mathbb{S}$ *is the type of link.*

The following function symbols are defined in this context:

- *Source(l) gives the source Source of link l*
- *Target(l) gives the target Target of link l*
- *Sort(l) gives the sort Sort of link l* $\square$

5.4.1 Object

An important modelling construct for creating a Behaviour Invocation Diagram is the concept of *object*. As discussed in chapter 3, in the VR-WISE approach, the Virtual Environment is populated by means of a number of *instances*. These instances are instantiations of *concepts* defined at the domain level. The object provides the link between the behaviour specifications and the static structure specifications. Using a BID, an instance of a concept can be given a number of behaviours (defined in a BDD). However it is also possible to connect behaviours to concepts itself. By assigning a behaviour to a concept, all instances of that concept will receive this behaviour. When a behaviour is assigned to an instance, only that particular instance will receive the behaviour. The assignment of a behaviour to an instance or to a concept is realized by assigning the actor for which the behaviour was defined to this instance or concept. Note that both a concept and an instance can have multiple actors assigned to it at the same time, meaning that all the behaviours from each of these actors are assigned to the concept or instance.

Definition 40 (Object). *Let BDD be a Behaviour Definition Diagram, $\boldsymbol{CONCEPT}$ be a set of concepts, $\boldsymbol{INSTANCE}$ be a set of instances (as defined in definition 3 and 4). An object o is defined as a pair $\langle$Entity, $\boldsymbol{ACTOR}\rangle$ where:*

- *Entity $\in \boldsymbol{CONCEPT} \cup \boldsymbol{INSTANCE}$ is either a concept or an instance,*
- *$\boldsymbol{ACTOR} \subseteq$ Actor(BDD) is a nonempty set of actors this object is referring to, and*
- *the following must hold: $\forall\ e \in \boldsymbol{ACTOR}$, $\forall\ p \in Property(e)$, $\exists!\ a \in Attribute(Entity)\ (Name(p) = Name(a) \land Unit(p) = Unit(a))$.* □

Concept AS Actor
Instance AS Actor
a
b

Figure 5.23: *Concept (a) and instance (b)*

In the graphical representation of a BID, a concept is drawn as a solid-line rectangle while an instance is drawn using an ellipse shape (see figure 5.23). The name of the object as well as the name(s) of the actor(s) that are assigned to the object are written inside the shape, as follows *name* ***AS*** *actor [, actor,...]*. If a list of objects, or a single item from a list of objects is to be represented, the same syntax is used as for the actor. Note that the objects in the Virtual Environment have a number of properties and these properties are those making an object compatible with a particular actor, meaning that the object can only be linked to an actor if it has the properties required by the actor(s). These properties are not depicted within the BIDs since they are defined in the Static Structure Diagram (SSD). Remember that a Static Structure Diagram is a diagram that is created during the Domain Specification or the World Specification (cfr. chapter 3), and thus contains the complete static structure of the Virtual Environment, i.e. the objects with their properties, relations and positions. Defining the properties here would result in redundancy in the conceptual models.

Entrance AS Door
a
Gate AS Sliding Door
b

Figure 5.24: *Example of concept (a) and instance (b)*

The examples shown in figure 5.24 illustrate the use of the object. In the first graphical notation (a), a concept called *Entrance* is assigned to the Door actor resulting in the fact that all instances of Entrance (e.g., of a store) will have the behaviours defined for the actor Door. In the second example (b), the instance called *Gate* is assigned to the Sliding Door actor meaning that only the Gate will have the behaviours that are defined for the Sliding Door actor.

5.4.2 Behaviour Reference

The second modelling concept in a behaviour invocation is the *behaviour reference*. The behaviour reference actually provides the link between the Behaviour Invocation Diagram and the Behaviour Definition Diagram. It can be seen as a sort of reference or pointer to a behaviour described in a Behaviour Definition Diagram. This element also gives the designer the possibility to parameterize the behaviour by giving concrete values to the parameters defined for the behaviour. Furthermore, a behaviour reference is also linked to the events that can be used to trigger the behaviour for the particular object. Events are discussed later on. Also note that a behaviour reference in the BID can be given a different name than the name of the behaviour to which it refers. Finally, a behaviour reference is also linked to the object(s) to which this behaviour belongs.

Definition 41 (Behaviour Reference). *Let BDD be a Behaviour Definition Diagram. A behaviour reference* $\mathtt{br}$ *is defined as a triple* $\langle$*Name, Behaviour,* $\boldsymbol{ATTRIBUTE}\rangle$ *where:*

- *Name* $\in \mathbb{S}$ *is the name of the behaviour reference,*

- *Behaviour* $\in$ *Behaviour(BDD) is the behaviour that this element is referring to,*
- $\boldsymbol{ATTRIBUTE}$ *is the set of attributes, representing the actual attributes for the behaviour Behaviour, and*
- *the following must hold:* $\forall\, a_{br} \in \boldsymbol{ATTRIBUTE}$, $\exists!\, a_{Behaviour} \in$ *Attribute(Behaviour)* $(Name(a_{Behaviour}) = Name(a_{br}) \wedge (Unit(a_{Behaviour}) = Unit(a_{br}))$.

The following function symbols are defined in this context:

- *Behaviour(br) is the behaviour Behaviour to which behaviour reference br is referring*
- *Attribute(br) gives the set of attributes* $\boldsymbol{ATTRIBUTE}$ *of the behaviour reference br* □

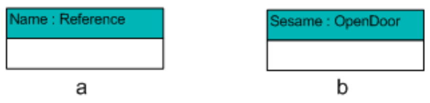


Figure 5.25: *Behaviour reference (a) and an example (b)*

The graphical notation for the behaviour reference is derived from the graphical notation of a behaviour used in the BDDs. It consists of a rectangle containing two compartments (see figure 5.25a). In the top compartment, the name of the behaviour is given as well as the name of the behaviour to which it refers using the syntax *name : reference*. The bottom compartment is again reserved for a script and will mostly be used to denote the values of the parameters required. A behaviour reference can have a number of events attached to it through a so-called *triggers link*. It can also have a number of objects attached to it through a so-called *from link*.

Figure 5.25b shows an example of a behaviour, called *Sesame*, referring to the *OpenDoor* behaviour that was defined earlier.

5.4.3 Causal Relation

Causal relations are used in Behaviour Invocation Diagrams to indicate relationships between behaviour references in a Behaviour Invocation Diagram. In fact, they are instantiations of relationships (implicitly or explicitly) defined between behaviours in a Behaviour Definition Diagram. As mentioned earlier, a composite behaviour may contain different sub-behaviours. This means that there is a relationship between the composite behaviour and each of its sub-behaviours. This is illustrated in figure 5.26a where A1 and A2 are sub-behaviours of A, A3 and A4 are sub-behaviours of A2, and B1 and B2 are sub-behaviours of B. Behaviours may also be related (connected to) explicitly by means of operators (e.g., temporal operators, lifetime operators,...). In figure 5.26a, an operator exists between A4 and B1, and between B and C. These implicit and explicit relationships are represented in a BID by means of *causal relations*. The relations derived from the decomposition of a composite behaviour into composite sub-behaviours are called *Use* relations (the behaviour uses the sub-behaviour to propertly execute). The relations derived from an explicit relationship between behaviours by means of an operator are called *Call* relations (the behaviour calls another behaviour in its execution). This is illustrated in figure 5.27b. These relationships and their associated (sub-)behaviours also need to be represented in a BID because these behaviours might also require parameters and might involve other actors than the ones attached to the composite behaviour. These parameters as well as these actors need to be specified.

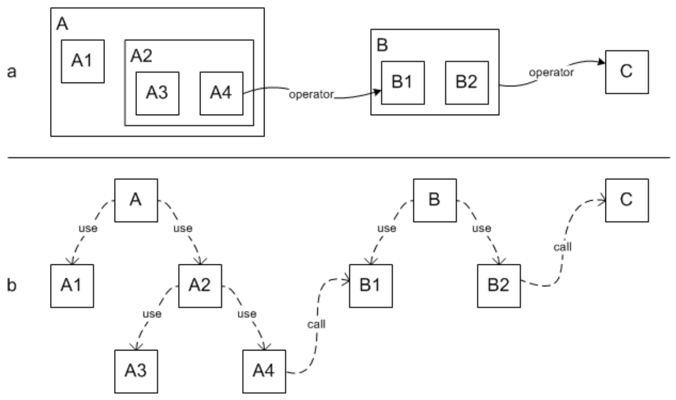


Figure 5.26: *Schematic Behaviour Definition Diagram (a), Behaviour Invocation Diagram (b)*

Definition 42 (Use Relation). *Let BDD be a Behaviour Definition Diagram. Let cb* $\in$ *Behaviour(BDD) be a composite behaviour in BDD and* $b_x \in$ *SubBehaviour(cb), then a Behaviour Invocation Diagram BID with behaviour references* br_s *and* br_t *where Behaviour(*br_s*) = cb and Behaviour(*br_t*) =* b_x*, should contain a link l, called use causal relation, where:*

- *Source(l) =* br_s *is the source of the link,*
- *Target(l) =* br_t *is the target of the link, and*
- *Sort(l) = "use".* □

Definition 43 (Call Relation). *Let BDD be a Behaviour Definition Diagram. Let c* $\in$ *Operator(BDD) be an operator in BDD where:*

- *Source(c), Target(c) ∈ Behaviour(BDD) is a behaviour in BDD,*

then a Behavior Invocation Diagram BID with behaviour references br_s and br_t where Behaviour(br_s) = Source(c) and Behaviour(br_t) = Target(c), should contain a link l, called call causal relation, where:

- *Source(l) = br_s is the source of the link,*
- *Target(l) = br_t is the target of the link, and*
- *Sort(l) = "call".* □

A causal relation is represented graphically as a dashed arrow from the dependent behaviour (the behaviour that is using or calling) to the dependending (the behaviour that is being used or called). The arrow has to be labelled with the type identifier (either "call" or "use") in angle qoutes. Figure 5.27a shows the notation for the *use relation* and Figure 5.27b shows the notation for the *call relation*.

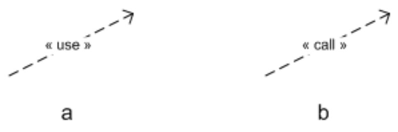


Figure 5.27: *Causal Relation: use (a) and call (b)*

5.4.4 Events

Using objects and behaviour references, we are able to specify the behaviour for the different objects. The only thing that is left to specify is when the behaviours should be invoked. In our approach, the *event* concept is used as a high-level modelling construct for specifying when behaviours should be triggered. There are different types of events supported in our approach: context events, time events, user events, and object events. We again define the concept of event in general first; next, the different types of events are explained and defined.

Definition 44 (Event). *An Event is either a Context Event, a Time Event, a User Event, or an Object Event.* □

We now discuss and define each type of event in more detail.

5.4.4.1 Context Event

The first type of event, the *context event*, enables the designer to specify the context (or situation) in which the behaviour of an object needs to be invoked e.g., when the temperature goes beyond 25 degrees Celsius. A context is defined as a condition on some entities. Entities are objects, users, or the environment in general, considered to be relevant for the behaviour in question. A simple context is defined using the following syntax *[property, subject, relater, value]*[4]. The property refers to the information this context is describing something about, subject is the user or any other object, or thing with which the context is concerned, value is the actual value associated with the property of the subject, and relater is a binary predicate that relates the property of the subject and the value. A relater can be a comparison operator (such as =, >, or <), or a self-defined predicate. Using simple contexts, more complex contexts can be constructed by combining contexts using the boolean operators AND, OR and NOT. An example of a context is *[temperature, Store, >, 25]*. This context could be used to for example open the windows when the temperature of the Store goes beyond 25 degrees Celsius.

Definition 45 (Context Event). *Let **CONCEPT** be a set of concepts, **INSTANCE** be a set of instances (as defined in definition 3 and 4). A context event* `se` *is defined as either a simple context event of a complex context event where:*

- *A simple context event* `sce` *is defined as a quadruple ⟨Property, Entity, Relater, Value⟩ where:*
  - *Entity* $\in$ ***CONCEPT*** $\cup$ ***INSTANCE*** *is the subject,*
  - *Property* $\in$ *Attribute(Entity) is the property of the subject,*
  - *Relater* $\in \mathbb{S}$ *is the relater predicate, and*
  - *Value* $\in \mathbb{S} \cup \mathbb{R}$ *is the value of the property.*
- *A complex context event* `cce` *is defined as a triple ⟨$Context_1$, $Context_2$, Join⟩ where $Context_1$, $Context_2$ are a simple or complex context events and Join* $\in$ *{ "AND", "OR", "NOT" } is the boolean operator combining the context events.* □

It is interesting to note that the Virtual Environment, the space on itself or the user, can be given as the subject of a context. This is done using the *Environment* or *User* identifier respectively. In our approach, a number of properties of the environment are predefined such as the gravity, temperature, humidity, pressure, and so on, which can be used as the information-item within the context. An example of such a context would be [humidity, Environment, =, 96]. This predefined list can easily be modified or extended by the designer by incorporating any domain-dependant environmental information.

[4]This structure should typically be interpreted as "the *property* of *subject* is *relater value*"

5.4.4.2 Time Event

A *time event* allows the designer to specify a moment in time on which the behaviour needs to be triggered. There are three different methods to specify the moment in time:

The first method is through a *relative time* specified in the following syntax *hh:mm:ss* (e.g., 1:30 meaning 1 minute 30 seconds) representing the time that has to pass counted from the start of the execution of the Virtual Environment. Note that not all the components of the relative time need to explicitely specified. For the ones that are omitted, a default is inferred. The result will be that the corresponding behaviour will be triggered when the prescribed amount of time has passed after starting the application.

The second method is to give an *absolute time* by means of a date-time through the following syntax *YYYY-MM-DD**T**hh:mm:ss* (e.g., 2007-02-07T15:19:30). Note that not all of the components of the absolute time need to be explicitly specified, some could be omitted and for those a default one is inferred. The result will be that when the specified moment in time is reached, an event is sent to the attached behaviour, which will then be triggered.

The third method is to give a more extended timing schedule in the form of a *recurrence pattern.* This is done using the syntax *duration [**FROM** begin [**TO** end]].* The duration indicates the time between two occurrences (the in-between time). An optional from-clause sets the starting date-time for the schedule while the to-clause sets the end date-time of the schedule. Every time the schedule is satisfied, an event will trigger the associated behaviour. For example: *1:30 **FROM** 13:00 **TO** 14:00* to indicate "every 1 minute 30 seconds between 1 PM and 2 PM", or *1:00:00 **FROM** 2007-04-15*, indicating "every hour from April 15th".

Definition 46 (Time Event). *Let **TIME** $\subset \mathbb{S}$ be the set of all formatted time strings. A time event* `te` *is defined as a pair ⟨Kind, DateTime⟩ where:*

- *Kind ∈ {"absolute", "relative", "recurrence"} is the type of event, and*
- *DateTime ∈ **TIME** is the string denoting the actual time(s) the event should fire.* □

5.4.4.3 User Event

Using a *user event*, the designer can specify that an action performed by the user, should be used as the triggering mechanism for the associated behaviour. The following user events are supported:

- *OnSelect*: The behaviour will be triggered when the main object of the behaviour (the object which is connected to the behaviour via a *from* *link*), is selected, i.e. when it is clicked with the mouse or selected through another selection technique.

- *OnTouch*: The behaviour will be triggered when the user has the mouse or any other pointing device over the main object.
- *OnVisible*: The behaviour will be triggered when the user can see a specific object as he navigates inside the world.
- *OnProxy(p)*: The behaviour will be triggered when the user has entered a particular perimeter p, given by a distance and a unit, around the object.
- *OnKeyPress(k,m)*: The behaviour will be triggered when a particular key-combination, given by a key k and a mask m, is pressed on the keyboard.

Next to these predefined user actions, also custom-made actions can be defined which allow behaviours to be triggered as a reaction on more complicated user interaction techniques (e.g., using menus, dialogs).

Definition 47 (User Event). *A user event* `ue` *is defined as a pair* $\langle$*Interaction,* ***ATTRIBUTE***$\rangle$ *where:*

- *Interaction* $\in$ *{"OnSelect", "OnTouch", "OnVisible", "OnProxy", "OnKeyPress"} is the type of interaction performed by the user, and*
- ***ATTRIBUTE*** *is the set of attributes (as defined earlier) passed to the event.* □

5.4.4.4 Object Events

The last type of event, the *object event* represents the event that is fired when two (or more) objects in the Virtual Environment interact with each other. Two types of object events are distinguished.

The *collision event* allows reacting to the situation where an object encounters an obstacle in the form of another object, i.e. when a collision between two objects occurs. This event needs to be further specified by means of a number of objects given as a parameter to the event. Note that the objects can be either concepts (meaning that a collision between all objects of the given concepts will be caught) or instances (meaning that only the collision between particular objects will be caught).

Definition 48 (Collision Event). *Let* ***CONCEPT*** *be a set of Concepts and* ***INSTANCE*** *be a set of Instances (as defined in definition 3 and 4). A collision event* `ce` *is defined as a monad* $\langle$***ENTITY***$\rangle$ *where* ***ENTITY*** $\subseteq$ ***CONCEPT*** $\cup$ ***INSTANCE*** *is the set of objects for which collision needs to be caught.* □

In addition, the *constraint event* allows to react when the limit of a particular constraint has been reached, or in the worst case, when a constraint has been violated. The *constraint* this event is referring to, is given by means of a first parameter. A second parameter is the set of named *limits* of the constraint that will fire the event and refer to the lower bound, upper bound or both.

Definition 49 (Constraint Event). *Let* $\boldsymbol{CONSTRAINT}$ *be a set of constraints (as defined in definition 6). A constraint event* $\mathtt{re}$ *is defined as a pair* $\langle Constraint, \boldsymbol{LIMIT}\rangle$ *where:*

- $Constraint \in \boldsymbol{CONSTRAINT}$ *is the constraint, and*
- $\boldsymbol{LIMIT} \subseteq Limit(Constraint)$ *is the set of limits to be taken into account for Constraint.* □

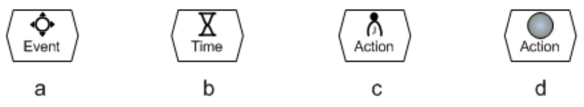


Figure 5.28: *Events*

An event is visualized in the diagram language as a flattened hexagon (see figure 5.28). The type of the event is indicated by an icon in the upper area. In the lower area, the additional information related to the event (such as the parameters) can be given. Figure 5.28a shows the graphical notation of a context event, figure 5.28b shows the notation of a time event, 5.28c gives the notation of a user event while figure 5.28d gives the representation of the object event.

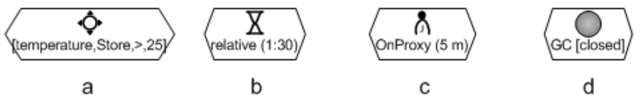


Figure 5.29: *Example of context event (a), time event (b), user event (c) and object event (d)*

In figure 5.29, examples are given. Figure 5.29a shows a context event in which a behaviour (for example OpenDoor) would be invoked when the temperature of the Store gets above 25 degrees Celsius. Figure 5.29b shows a relative time event. In this case, the behaviour linked to this event would be triggered 1 minute 30 seconds after the world has started to exist. Figure 5.29c shows a user event. The behaviour that would be attached to this event would be invoked when the user (represented by an avatar or virtual pointer) comes into the vicinity of the object, more specifically within a distance of 5 meters. Figure 5.29d shows the constraint event where the behaviour attached to this event would be triggered when the constraint on the gate, GateConstraint, is in its closed limit.

5.4.5 Example

This section will now illustrate the use of a Behaviour Invocation Diagrams and the modelling concepts introduced in the previous sections. In section 5.3.5 on page 126, an example was given of a virtual robot in a manufacturing plant for which a Behaviour Definition Diagram was made. This section will build further on the same example (see figure 5.30).

The Master actor is assigned to the Assembler instance. Hence, the Assembler will possess the behaviour defined for the Master actor, i.e. ProductionCycle. This behaviour can be triggered by means of a user-event, namely OnClick. This means that when the user clicks on the Assembler this behaviour will be executed. When the ProductionCycle behaviour is attached to an object, also the behaviours that are causally linked (represented by the dashed arrows) to this behaviour need to be assigned to objects. To assign behaviour to Gripper A and Gripper B, the actors RightArm, respectively LeftArm are assigned to them. For the actors RightArm and LeftArm the behaviour RightArmMovement, respectively LeftArmMovement were defined (not shown here). These two behaviours are triggered by the ProductionCycle. Therefore, there are no events needed to trigger these behaviours. The Switch instance is given behaviour by assigning the Control actor to it. As a consequence, this instance will have two behaviours PowerOn and PowerOff. Both the PowerOn and the PowerOff behaviour must be invoked by a time-event (at 9h00 for PowerOn and at 18h00 for the PowerOff).

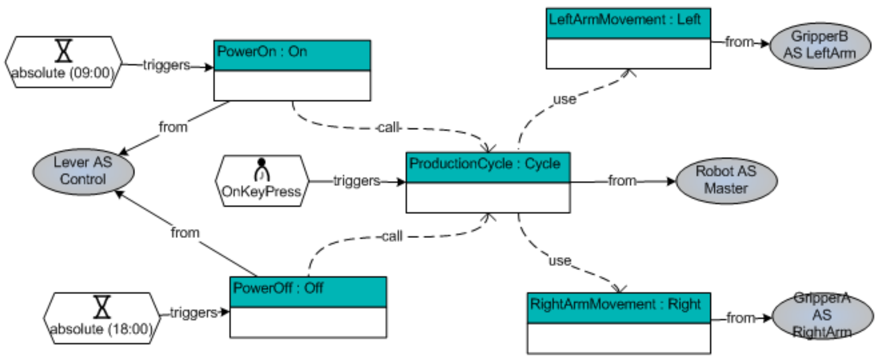


Figure 5.30: *Behaviour Invocation Diagram: Virtual Robot*

5.5 Behavioural Script Language

In the first two sections of this chapter, the Behaviour Definition Diagram and the Behaviour Invocation Diagram were introduced. These diagrams, and the corresponding modelling elements that were defined for them, are sufficient to specify basic behaviours for the objects in the Virtual Environment. Until now, the behaviour modelling language was put forward as a purely diagrammatical language. The advantage of a diagrammatical language is its ease of use, which makes it popular amongst non-programmers and suitable to communicate the design with non-technical people. However, graphical languages are in general less expressive than a textual (scripting) language. This is also the case for the graphical behaviour modelling language presented. In other words, it is not (yet) possible to specify behaviours with a high degree of complexity using only the graphical notation. Therefore, the behaviour modelling language was extended and turned into a hybrid mix of a graphical language and a textual language [Pellens et al., 2006b]. Issues that are best expressed graphically are expressed graphically, whereas issues that are best expressed textually are done textually through an intuitive scripting language.

Therefore, a high-level specification language, called Behavioural Script Language (BSL), is defined. It aims at providing the designer a complementary formalism for specifying behaviours that are more complex. This language allows the designer to parameterize behaviours, and to specify additional (optional) modifiers for the different actions that were defined earlier in this chapter. Furthermore, conditions can be specified which may either prevent or allow the actions to be executed. Finally, also scripts can be created which should be executed before, during and/or after the execution of the behaviours. Please note that the scripting language is not used in any way for combining different behaviours to each other in order to create a complex behaviour. It is only used in order to further specify the behaviour (action) with more details and thereby making the behaviour (action) itself a more complex one.

In this work, the specification of an intuitive, expressive and specific domain programming language was preferred over an existing (high-level) general purpose programming language (e.g., Java). Note, that the problem at hand (i.e. offering the designer the possibility to complete the object behaviour with more detailed specifications) is a very specific one, and for a very specific domain (i.e. the domain of Virtual Reality). Such a case is best supported by a *domain-specific language*. The field of domain-specific languages aims at constructing designated languages for different domains and from which the instructions are closely related to the terminology of the domain. The scripting language that is used results in specifications that are more oriented towards the behaviour modelling domain, which may increase the possibility that both programmers and non-programmers can easily use the language.

In this sense, BSL can be called a domain-specific language. BSL aims to keep the middle between the expressiveness of a generic (high-level) programming language and the specificity of a domain-specific language. However, it does not offer all of the expressive power usually found in higher-level general purpose programming languages. One goal was to keep the learning time as short as possible, also for non-programmers.

This section describes the lexis, the syntax, and the semantics of the Behavioural Script Language (BSL). In the end of this section, a few examples are given about the use of the scripting language in the graphical notation.

The language constructs from BSL will be explained using the extended Backus Naur Form [Backus et al., 1960]. The following conventions are used in the remaining part of this section:

- Curly brackets {a} are used for repetitions, meaning 0 or more a's.
- Squared brackets [a] are used for optional elements.
- The keywords are enclosed in single quotes and cannot be used as identifier elsewhere.
- Terminal symbols are shown in `typewriter` font and enclosed in single quotes.

5.5.1 General Overview

Before going into the structure of a script defined in the BSL, some general production rules used in the remaining part of this chapter are firstly introduced.

Identifiers are names that the designers can choose for their functions, variables, etc. An *identifier* must be a whole word, made up of characters or digits but starting with a character. The standard keywords are reserved and cannot be used as identifier. This corresponds with the definition of names in most programming languages. The definitions of a character and a digit are deferred until later, but for the moment it can be assumed that they are the same ones as used in the English language. An identifier can be further qualified into an *identifier expression* (id-exp) by means of the dot (".") notation. A *list of identifiers* is generally separated by means of commas.

⟨*identifier*⟩ ::= ⟨*character*⟩ { (⟨*character*⟩ | ⟨*digit*⟩) }

⟨*id-exp*⟩ ::= ⟨*identifier*⟩
    | ⟨*id-exp*⟩ '.' ⟨*identifier*⟩

⟨*id-list*⟩ ::= ⟨*identifier*⟩ { ',' ⟨*identifier*⟩ }

A script in BSL consists of a number of entries. The beginning of an entry is denoted with a slash ('/') together with the name of the entry and ends when the following entry is encountered or when the script is finished. All entries are optional and can only be used once. An empty script is a valid script. Note that the examples until now all had empty scripts. The scripts only apply to the behaviour (action) in which they are specified. In the following sub-sections, each of the different possible entries will be discussed.

⟨*script*⟩ ::= ['/speed' ⟨*speed-declaration*⟩]
    ['/speedtype' ⟨*speedtype-declaration*⟩]
    ['/repeat' ⟨*repeat-declaration*⟩]
    ['/variables' ⟨*variables-declaration*⟩]
    ['/condition' ⟨*condition-declaration*⟩]
    ['/before' ⟨*block*⟩]
    ['/do' ⟨*block*⟩]
    ['/after' ⟨*block*⟩]

5.5.2 Speed

The *speed* entry specifies the rate at which the behaviour should be executed. The possible values for this flag range from *very slow*, *slow*, *normal*, *fast* to *very fast*. Note that the speed is not specified in exact values but rather using the intuitive counterparts which can be (re)set in the environment in general.

⟨*speed-declaration*⟩ ::= ⟨*velocity*⟩
    | ⟨*velocity*⟩ 'for' ⟨*id-list*⟩ { ';' ⟨*velocity*⟩ 'for' ⟨*id-list*⟩ }

⟨*velocity*⟩ ::= '`very slow`' | '`slow`' | '`normal`' | '`fast`' | '`very fast`'

Since some behaviours involve more than one object, the designer needs to have the flexibility to control the speed of all the objects independently instead of one speed for all of them (which is the default). This can be achieved by specifying a specific actor (or list of actors) using the for-variant.

Some examples of setting the speed are:

- **/speed** 'very slow'
- **/speed** 'fast' **for** Car

5.5.3 Speedtype

The *speedtype* entry specifies how a movement needs to be executed: one can accelerate gradually and decelerate gradually (*accelerate/decelerate*), a movement at a constant speed (*constant*) can be used, or the deceleration (*decelerate*) respectively acceleration (*accelerate*) can be specified. Also here, note the use of intuitive terms to indicate the type of movement performed.

⟨*speedtype-declaration*⟩ ::= ⟨*type*⟩
   | ⟨*type*⟩ 'for' ⟨*id-list*⟩ { ';' ⟨*type*⟩ 'for' ⟨*id-list*⟩ }

⟨*type*⟩ ::= '`decelerate`' | '`accelerate`' | '`constant`' | '`accelerate/decelerate`'

In the same way as it was the case for the speed, different speedtypes can be specified for the different objects involved in a behaviour.

Examples of using the speedtype element are:

- **/speedtype** 'accelerate/decelerate'
- **/speedtype** 'constant' **for** Car ; 'accelerate' **for** Motorbike

5.5.4 Repeat

The *repeat* entry is used to denote the number of times that the behaviour needs to be repeated (executed). This is given either by means of a constant value or by means of a variable (variables will be defined next).

⟨*repeat-declaration*⟩ ::= (⟨*numeral*⟩ | ⟨*variable*⟩) 'time(s)'

An example of the repeat entry is shown below:

- **/repeat** 5 **time(s)**
- **/repeat** x **time(s)**

5.5.5 Variables

The *variables* entry allows the designer to use variables for a particular behaviour. Variables are placeholders for different values. We use an un-typed system for variables, so variables can hold values of any type. This facilitates the use of variables for non-programmers because those people are usually not familiar with types in the way programmers are used to. There are actually two kinds of variables in the BSL: global variables and local variables. The global variables are those that are defined for the environment and which are not defined within a behaviour. Global variables can be referenced in the behaviours and their value can be changed by the behaviours. The local variables are those that are defined within a behaviour. All variables must be assigned a value before they can be used. A constant or an expression can be assigned to a variable in the following way:

⟨*variables-declaration*⟩ ::= ⟨*assignment*⟩ { ';' ⟨*assignment*⟩ }

⟨*assignment*⟩ ::= 'assign' ⟨*exp*⟩ 'to' ⟨*variable*⟩

⟨*variable*⟩ ::= ⟨*identifier*⟩
  | 'ThisAction' '.' ⟨*identifier*⟩

The variable itself is identified by an identifier. It can optionally be further qualified by the ThisAction keyword (similar to 'this' or 'self' in programming languages) in case there should be any name conflicts with the expression that is assigned to this variable.

A few examples of setting up new variables are given:

- **/variable assign** 5 **to** x
- **/variable assign** 5 **to** x ; **assign** x+5 **to** y

As mentioned before, a number of predefined variables that have constant values are available in our approach. These constants obviously cannot be changed by the designer. These constants are PI (pi), Natural Logarithm (e), Frame Number (F). Another important variable is Time which gives the current absolute time. The above variables are constants belonging to the environment and are preceded by the ThisEnvironment qualifier. Other variables are ElapsedTime giving the time that has elapsed since the behavior has been executed or triggered, and the LoopTime which provides the amount of seconds since this behavior was last executed. These last variables are belonging to a behaviour and must be preceded by the ThisAction qualifier.

5.5.6 Conditions

Whether a behaviour will be executed or not is determined by the value of the conditional expression that can be specified for this behaviour. A *condition* must result in a true or false value. If the value is true, the behaviour will be executed, otherwise the execution will not (re)start. In the latter case, the execution proceeds with the following behaviour. The condition is defined as a conditional expression in which basically everything from a normal expression can be used, except an assignment. A normal expression will be explained further down.

⟨*condition-declaration*⟩ ::= ⟨*conditional-exp*⟩

⟨*conditional-exp*⟩ ::= ⟨*disjunctive-exp*⟩

The use of conditions within a behaviour can be done as follows:

- **/condition** temperature >= 24
- **/condition** (code == "7893") **and** (**not** open)

The difference between this conditional expression and the conditional operator that was discussed earlier is the fact that the conditional operator is used in a case where a kind of branching semantics is required (e.g., where either one behaviour or the other is executed). The conditional expression is mostly used in combination with the repeat entry to construct an iteration (e.g., execute the behaviour while the condition is true).

5.5.7 Before, Do and After Blocks

The *before* entry allows the designer to specify statements, specified as a block-statement, that need to be executed before the actual behaviour is executed. The *after* entry is similar to the before entry but allows to specify a block-statement that needs to be performed once the behaviour has been completed. The *do* entry identifies an ongoing activity that is performed as long as the behaviour is in the active state or until the computation specified by the block-statement is completed.

All three entries allow performing some calculations and/or set the values of variables needed in the behaviour and hereby, allow customizing behaviours.

The unit of execution in BSL is called a block. A block is a sequence of statements, which are executed sequentially. The different kinds of statements available in the BSL are given below. All of them will be further explained.

⟨*block*⟩ ::= { ⟨*statement*⟩ }

⟨*statement*⟩ ::= ⟨*assignment*⟩
  | ⟨*function-call*⟩
  | ⟨*returning-exp*⟩
  | 'increment' ⟨*postfix-exp*⟩ ['by' ⟨*exp*⟩]
  | 'decrement' ⟨*postfix-exp*⟩ ['by' ⟨*exp*⟩]
  | 'while' ⟨*exp*⟩ 'do' ⟨*block*⟩ 'end'
  | 'when' ⟨*exp*⟩ 'do' ⟨*block*⟩ ['otherwise' ⟨*block*⟩] 'end'
  | 'for' ⟨*identifier*⟩ 'from' ⟨*exp*⟩ 'to' ⟨*exp*⟩ ['by' ⟨*exp*⟩] 'do' ⟨*block*⟩ 'end'

5.5.7.1 Assignment

A first type of statement is the *assignment* which was already discussed above. It serves the purpose of defining new local variables (local to the block in which they are defined) or update variables. The variables that can be updated are either local variables of the behaviour, or the global variables of the environment itself. In the latter case, they must be preceded by the *ThisEnvironment* keyword.

A few examples of assigning values to variables are given here:

- **assign** 25 **to** ThisEnvironment.temperature

- **assign** 5 **to** index
- **assign** counter **to** ThisAction.index

5.5.7.2 Function-call and Function-definition

The second type of statement is the *function-call.* The function-call allows calling (or executing) a particular function, or piece of code defined somewhere else. A call is made through the name of the function together with zero or more expressions being the arguments of the function. A number of standard mathematical functions are predefined by the approach (such as sine, cosine, tangent, squareRoot, and so on). Furthermore, some other functions related with time are available: *getTime* provides the current (absolute) time; *elapsedTime* gives the time that has elapsed since the behaviour has been started, and *loopTime* returns the amount of time (seconds) since this behaviour was last executed. Additional functions to retrieve information from the actors (or the user) are also defined: *getPosition* returns the location of the actor (or user); and *getOrientation* returns the orientation of the actor (or user).

⟨*function-call*⟩ ::= 'invoke' ⟨*function-name*⟩ ['with' '(' [⟨*explist*⟩] ')']

⟨*function-name*⟩ ::= ⟨*id-exp*⟩

⟨*explist*⟩ ::= { ⟨*exp*⟩ ',' } ⟨*exp*⟩

Please note that next to the calling of native functions provided by the system, a designer can also define his own functions (externally) and call these functions in the behaviours. A function is identified by an identifier, and has zero or more parameters, which on their turn are identified by identifiers. When defining a custom function, the return statement can be used to return any result coming from the function.

⟨*function*⟩ ::= 'define' ⟨*function-name*⟩ ['with' '(' [⟨*parlist*⟩] ')'] ⟨*block*⟩ 'end'

⟨*parlist*⟩ ::= ⟨*identifier*⟩ {',' ⟨*identifier*⟩ }

⟨*returning-exp*⟩ ::= 'return' ⟨*exp*⟩

Using externally defined *functions*, the designer (or somebody else, e.g., a more experienced programmer) has the possibility to define some frequently used higher-level operations once and then, use them several times. Indeed, using functions, higher-level operations on the relevant actions can be defined using the BSL language elements discussed in this chapter. In this way, more intuitive (composed) operations can be defined, which makes the

BSL more intuitive to use for a non-programmer. Furthermore, functions can also be used to reduce the length of a script. Moreover, the functions are independent of any particular behaviour, and thus can be used in combination with different behaviours. In this way, libraries of functions can be shared among designers.

Some examples of function-call and function-definitions are:

- **invoke** sine **with** (angle)
- **define** rollingMotion **with** (xco, radius) return xco/radius **end**
- **invoke** rollingMotion **with** (x, r)

5.5.7.3 Increment and Decrement

Two commonly used statements are the *increment* and the *decrement*, which serve to increase (or decrease) the value of a variable by a specified amount. If no amount (to increment or decrement) is specified then the default (1) will be taken.

The syntax for the increment respectively decrement is given as follows:

⟨*statement*⟩ ::= 'increment' ⟨*postfix-exp*⟩ ['by' ⟨*exp*⟩]
| 'decrement' ⟨*postfix-exp*⟩ ['by' ⟨*exp*⟩]

The increment and decrement are used as follows:

- **increment** i **by** 2
- **decrement** j

5.5.7.4 Control statements

The last three rules of the statement refer to the basic control structures. BSL has some intuitive ways to conditionally control the flow of the execution.

- A *when* statement executes code depending on whether a boolean expression is true. This statement consists of two different parts, one of it is optional. The first statement block is executed if the expression evaluates to true and otherwise, the second statement block (if it is specified) is executed.

  ⟨*statement*⟩ ::= 'when' ⟨*exp*⟩ 'do' ⟨*block*⟩ ['otherwise' ⟨*block*⟩] 'end'

- The *while* loop repeatedly executes a statement block as long as a boolean expression is true. It contains two parts, the expression which is tested before the second part namely the statement block, is executed.

⟨*statement*⟩ ::= 'while' ⟨*exp*⟩ 'do' ⟨*block*⟩ 'end'

- The *for* loop can be more convenient than a while loop when you need to maintain an iterator. It contains five parts, of which one is optional. The first part is an identifier used as a local iterator variable; the second part is the lower-limit; the third part is the upper-limit while the fourth part is the step value. The loop is repeated for the variable starting at the value of the lower-limit, until it passes the upper-limit by steps of the step value.

  ⟨*statement*⟩ ::= 'for' ⟨*identifier*⟩ 'from' ⟨*exp*⟩ 'to' ⟨*exp*⟩ ['by' ⟨*exp*⟩] 'do' ⟨*block*⟩ 'end'

Examples of the different control structures are listed here:

- **when** temperature >= 25 **do assign** 'very fast' **to** speed **end**
- **while** (x >= 25) and (x <= 30) **do increment** temperature **by** 1 **end**
- **for** it **from** 1 **to** 3 **do assign** it * 15 **to** distance **end**

5.5.8 Remaining rules

The remaining EBNF production rules are discussed in this section. Firstly, the production rules involved in defining an expression are discussed (which was already encountered many times above). Afterwards, the low-level rules for specifying the basic data types are discussed.

5.5.8.1 Expressions

The logical operators in BSL are **or**, **and** and **not**. Equality (==) compares the type of its operands. If the types are different, then the result is false. Otherwise, the values of the operands are compared. Numbers and strings are compared in the usual way. The operator not equals (<>) is exactly the negation of equality (==). Furthermore, it supports the basic relational operators such as < (smaller than), > (greater than), <= (smaller than or equals) and >= (greater than or equals). BSL supports the usual arithmetic operators: the binary **+** (addition), **-** (subtraction), ***** (multiplication), **/** (division), **%** (modulo), and unary **-** (negation). The precedence of the operators is similar to those of an ordinary scripting language and encoded through the BNF structure (see below). As usual, parentheses can be used to change the precedence of an expression.

⟨*exp*⟩ ::= ⟨*assignment-exp*⟩

⟨*assignment-exp*⟩ ::= ⟨*disjunctive-exp*⟩ | ⟨*assignment*⟩

⟨*assignment*⟩ ::= 'assign' ⟨*exp*⟩ 'to' ⟨*variable*⟩

⟨*disjunctive-exp*⟩ ::= ⟨*conjunctive-exp*⟩
  | ⟨*disjunctive-exp*⟩ 'or' ⟨*conjunctive-exp*⟩

⟨*conjunctive-exp*⟩ ::= ⟨*equality-exp*⟩
  | ⟨*conjunctive-exp*⟩ 'and' ⟨*equality-exp*⟩

⟨*equality-exp*⟩ ::= ⟨*relational-exp*⟩
  | ⟨*equality-exp*⟩ ('==' | '<>') ⟨*relational-exp*⟩

⟨*relational-exp*⟩ ::= ⟨*additive-exp*⟩
  | ⟨*relational-exp*⟩ ('<' | '>' | '<=' | '>=') ⟨*additive-exp*⟩

⟨*additive-exp*⟩ ::= ⟨*multiplicative-exp*⟩
  | ⟨*additive-exp*⟩ ('+' | '-') ⟨*multiplicative-exp*⟩

⟨*multiplicative-exp*⟩ ::= ⟨*unary-exp*⟩
  | ⟨*multiplicative-exp*⟩ ('*' | '/' | '%') ⟨*unary-exp*⟩

⟨*unary-exp*⟩ ::= 'increment' ⟨*unary-exp*⟩
  | 'decrement' ⟨*unary-exp*⟩
  | ⟨*postfix-exp*⟩
  | ('+' | '-') ⟨*unary-exp*⟩
  | ⟨*negative-exp*⟩

⟨*negative-exp*⟩ ::= 'not' ⟨*unary-exp*⟩
  | ⟨*postfix-exp*⟩

⟨*postfix-exp*⟩ ::= ⟨*primary*⟩ | ⟨*id-exp*⟩

⟨*primary*⟩ ::= ⟨*literal*⟩ | '(' ⟨*exp*⟩ ')' | ⟨*function-call*⟩ | ⟨*attribute-access*⟩

⟨*attribute-access*⟩ ::= 'ThisAction' '.' ⟨*identifier*⟩
  | 'ThisEnvironment' '.' ⟨*identifier*⟩

Here are some examples of expressions:

- (distance + threshold) <= perimeter

- ThisAction.pressure < ThisEnvironment.pressure

- 2 * PI / 3

- -1.5734

- true

5.5.8.2 Basic Data Types

A *literal* is the most primitive kind of expression in BSL. A literal can be interpreted in four different ways. A *numeral* consists of a numerical constant, not starting with a 0, and which may be written with an optional decimal part. A *boolean* is a type containing fixed values that can be either *true* or *false*. A *string* is a series of characters or digits, delimited by matching double quotes, while a *character* is delimited by matching single quotes.

⟨*literal*⟩ ::= ⟨*numeral*⟩ | ⟨*boolean*⟩ | ⟨*string*⟩ | ⟨*character*⟩

⟨*boolean*⟩ ::= ’`true`’ | ’`false`’

⟨*numeral*⟩ ::= [’`-`’] ⟨*number*⟩ { ⟨*digit*⟩ } [’.’ ⟨*digit*⟩ { ⟨*digit*⟩ }]
  | [’`-`’] ’`0`’ [’.’ ⟨*digit*⟩ { ⟨*digit*⟩ }]

⟨*string*⟩ ::= ’`"`’ { (⟨*character*⟩ | ⟨*digit*⟩) - ’`"`’ } ’`"`’

⟨*character*⟩ ::= ”’ (⟨*character*⟩ | ⟨*digit*⟩) - ”’ ”’

⟨*character*⟩ ::= ’`a`’ | ’`b`’ | ’`c`’ | ’`d`’ | ’`e`’ | ’`f`’ | ’`g`’ | ’`h`’ | ’`i`’ | ’`j`’ | ’`k`’ | ’`l`’ | ’`m`’ |
  ’`n`’ | ’`o`’ | ’`p`’ | ’`q`’ | ’`r`’ | ’`s`’ | ’`t`’ | ’`u`’ | ’`v`’ | ’`w`’ | ’`x`’ | ’`y`’ | ’`z`’ | ’`A`’ | ’`B`’ | ’`C`’
  | ’`D`’ | ’`E`’ | ’`F`’ | ’`G`’ | ’`H`’ | ’`I`’ | ’`J`’ | ’`K`’ | ’`L`’ | ’`M`’ | ’`N`’ | ’`O`’ | ’`P`’ | ’`Q`’ | ’`R`’ |
  ’`S`’ | ’`T`’ | ’`U`’ | ’`V`’ | ’`W`’ | ’`X`’ | ’`Y`’ | ’`Z`’

⟨*digit*⟩ ::= ’`0`’ | ⟨*number*⟩

⟨*number*⟩ ::= ’`1`’ | ’`2`’ | ’`3`’ | ’`4`’ | ’`5`’ | ’`6`’ | ’`7`’ | ’`8`’ | ’`9`’

As mentioned before, the scripts that have been discussed in this section can be used in the graphical notation to turn the behaviours (actions) into more complex behaviours (actions). Figure 5.31 gives two examples of how the scripting language can be applied. Figure 5.31a presents a move action where an additional *speed* and *speedtype* have been defined stating that the object needs to move forward but in a very slow manner and accelerating in the process. Figure 5.31b presents a composite behaviour called *PerformAction* which should be executed three times. Two variables have been defined, namely *i* and *d*. Before every execution, the *d* variable is modified and after every execution, the *i* variable is incremented.

5.6 Summary

This chapter has described the first major contribution of this dissertation. Within this chapter, our Behaviour Modelling Language (BML) is explained in detail. The language is a combination of a graphical language and a textual scripting language. The BML is part of the overall VR-WISE approach

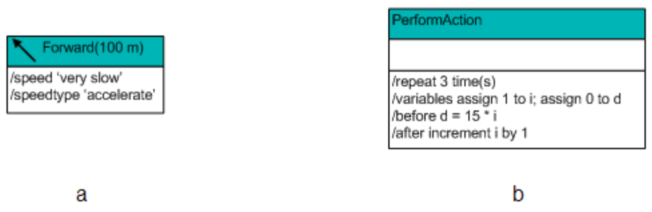


Figure 5.31: *Example use of script in graphical notation*

described in chapter 3. Three different types of diagrams were introduced to fully describe object behaviour in a Virtual Environment. These diagrams are the Behaviour Definition Diagram, the Behaviour Invocation Diagram and the Structure Chunk. The chapter furthermore described the notation and explained the semantics of the different language elements developed for these diagrams.

The Behaviour Definition Diagram (BDD) allows the designer to define behaviours. A BDD is represented as a graph-like diagram. The BDD consists of a number of actors. An actor represents an object that is involved in a behaviour. Actors are connected to behaviour elements, which describe how the objects can behave. Behaviours can be combined through a set of operators to form composite behaviours.

In different places inside a BDD, Structure Chunks (CS), describing the static structure of a set of objects at a particular moment in time, are used. These are graphs containing items connected to each other by means of relations.

The Behaviour Invocation Diagram (BID) is also represented as a graph in the diagrammatical notation. A BID consists of a number of actual objects populating the Virtual Environment. These objects play the roles of actors and thereby receive the behaviours, referred to by behaviour references, defined for the actors in the BDDs. Furthermore, the behaviour references are linked with events to specify how the behaviours can be invoked.

A graphical language has many advantages. Diagrams help the user to quickly build a mental model as the information is presented more explicitly than in case of textual languages. Diagrams also allow hiding details when not relevant. However, since a purely graphical notation was not enough to fully describe complex behaviours, the graphical language has evolved into a mixed graphical/textual language. Therefore, the chapter also defines the Behavioural Script Language (BSL), which is the textual part of the BML. The purpose of BSL is to give the designer the possibility to enhance the behaviours with scripts and in this way allow specifying behaviours that could not have been (easily) described using only a graphical notation.

CHAPTER 6

Behavioural Design Patterns Framework

The previous two chapters discussed the behaviour modelling approach that was added to the general VR-WISE approach. They explained in depth the different steps involved in the process of modelling the behaviours as well as all the high-level modelling concepts available for doing so. Also the graphical notation that was developed for this approach was introduced. This chapter will further extend the behaviour modelling approach by combining the concept of behavioural design patterns with the graphical notation described earlier. In this way, the modelling of more complex behaviours is facilitated and the accessibility towards laymen may be improved.

This chapter is structured as follows. First, in section 6.1, some observations from either the related work that was investigated and from our own experience obtained with the graphical notation developed in previous chapters, are reviewed. This leads us to section 6.2 where the concept of generative design patterns is introduced as a mechanism to overcome some of the issues concerning the observations made earlier. In section 6.3, the chapter then describes a behaviour modelling framework combining our graphical notation with generative design patterns. The framework allows more experienced designers to create their own patterns and make them available in our graphical notation. These patterns can then be used by both experienced and non-experienced designers in the behaviour specifications. Next to the behavioural pattern framework, the chapter also describes (in section 6.4) a number of patterns developed within this framework. Finally, section 6.5 will give a short discussion about this chapter.

6.1 Observations

The graphical notation, introduced in the previous chapter, has been employed in a number of use cases (see chapter 8). The experience that was gained from using our behaviour modelling approach on the one hand, and by investigating the current state-of-the-art [Pellens et al., 2006a] in behaviour modelling in general on the other hand, has resulted in a number of observations. These observations apply to the work that was set out earlier in this dissertation but can also be generalized to other graphical notations as well.

A first observation is that graphical notations do not scale well [Whitley, 1997]. The graphical models (i.e. diagrams) can become very complex for more complex situations, up to the point where they become unmanageable. Furthermore, scripting languages also have their limitations and/or drawbacks. The expressiveness of a graphical notation is usually limited compared to a textual language, especially if one wants to keep the number of graphical elements low. An in-depth evaluation of the related work did reveal that the problem faced by these approaches was that either the diagrams or the code are becoming too large when specifying complex behaviour, resulting in specifications that are not easy to read and to maintain. As our approach is based on a mixed graphical-textual notation, it is also facing this kind of problems when modelling complex behaviours. Another issue is that for some range of specifications, optimized implementations might already exist. This requires some kind of mechanism to reuse those existing implementations as a kind of black box rather than to re-specify them from scratch.

A second observation is that in practice many people using specialized VR tools (and forced to learn the associated scripting language) are trying to avoid developing long scripts by using and adapting existing scripts, either built-in scripts or scripts coming from external sources. An interesting study has been performed by McNaughton et al.. In [McNaughton et al., 2004a], the authors investigated how easily non-programmers are using a scripting language for specifying behaviours in the case of a game engine. This study showed that non-programmers often reuse existing scripts as they do not have enough knowledge about the scripting language itself and on how to specify behaviour in general. For this reason, they often try to find a script defining a behaviour close to the one they want to model, so that they can modify it to fit their needs. This study also showed that people without programming skills spend a lot of time sending questions over mailing lists in order to be able to understand the script they have found. Part of the reason for this is that variables used in a script have a name from which the meaning cannot be derived intuitively. The situation becomes even worse when multiple (pieces of) scripts need to be combined to fulfill a particular task. Often not all the necessary changes are made, or some of the changes are done incorrectly leading either to non-correct scripts or to wrong behaviour.

6.2 Visual Generative Design Patterns

Based on the observations made above, we can conclude the following. Firstly, the design of more complex behaviour needs to be facilitated better and secondly, a mechanism which will allow to (re)use existing solutions to particular problems encountered when modelling behaviour in Virtual Environments needs to be provided. To achieve this, our approach has been extended with the concept of "design patterns".

The concept of design pattern was first introduced by Christopher Alexander [Alexander et al., 1977]. However, the patterns that he introduced did not refer at all to design patterns in computer science. Instead, it was about architecture and the process of building successful buildings and towns. He documented the best practices of building and showed that patterns are a good way to capture the wisdom of a craft.

The work of Alexander has greatly influenced the computer science domain and in particular the Software Engineering domain where a lot of research has been performed in the context of design patterns for software applications [Schmidt, 1995]. As a result, a number of best practices for software design involving design patterns have been published (e.g., [Gamma et al., 1995]). A design pattern specifies, in a systematic and general way, a well-defined solution to design problems that often appear when designing and developing software.

The use of design patterns, which can be applied to the design of Virtual Environments as well, offers many advantages. They can be stated as follows:

- **Capture of expertise.** The patterns are usually designed by more experienced people. Once they exist, it gives a mechanism to less experienced people to exploit the knowledge and expertise of the pattern designer in a well-defined way.

- **Communication improvement.** The use of design patterns will improve the communication between the different stakeholders (i.e. both non-technical people and technical people) of the application because it allows abstracting from too much complexity.

- **Higher reusability.** The patterns promote reuse (of existing algorithms and data structures) since a pattern can be designed and implemented once and then be instantiated many times in different Virtual Environments, maybe each time in a slightly adapted way.

- **Faster development.** The time of development can be reduced significantly by providing a library of commonly used patterns, allowing the designer to select his pattern, fill in the details and make some adaptations.

In Software Engineering, the patterns are in most cases documented through a text description, using a fixed template. However, in this way there is a big disadvantage that still remains, i.e. the patterns must be manually implemented (programmed) each time they are applied. For non-programmers, the implementation of a pattern may be difficult and error-prone. As a way to solve this, MacDonald et al. have introduced the concept of *Generative Design Patterns* [MacDonald et al., 2002]. Generative design patterns are similar to regular design patterns except that they are defined in such a way that it becomes possible to automatically generate code from them. They remove the burden of implementing the patterns from the user. They offer a number of advantages besides the ones offered by traditional design patterns. These advantages are:

- **Code generation.** Many people are unable to write scripting code themselves and therefore, they must rely on experienced programmers. The generative design patterns for which the code can be automatically generated enable the user to incorporate larger pieces of code into their Virtual Environment without having to program a single line of code.

- **Fewer errors.** Since the design patterns can be used as black boxes and the code can be automatically generated for them, errors can be avoided which would otherwise arise when scripts are copied and modified manually.

In the remaining part of this chapter, the concept of generative design patterns is used for specifying patterns of behaviours that often occur in Virtual Environments. These generative design patterns are then turned into what are called *Visual Generative Design Patterns* by expressing them formally using a graphical notation and providing a mechanism to generate code when these patterns are used. Furthermore, it will be shown how these visual generative design patterns are created using a framework that allows them to be integrated into our existing graphical behaviour modelling language. The combination of the two paradigms, generative design patterns and graphical languages, allows us to cope with the problems of scripting, namely not being accessible to non-programmers. At the same time, we can keep the advantages of a graphical language, namely the ease-of-use and reduction in programming errors. All this can be achieved while still being able to properly represent more complex behaviour and simultaneously capture expertise from others by means of the behavioural design patterns.

Note that in order to make the generative design patterns visual, the framework that is described here employs our graphical notation as described above. However, a different graphical notation could have been taken instead and this approach would still be applicable. In other words, the approach discussed in this chapter is independent of the graphical notation used.

6.3 The Design Patterns Framework

In this section, the details on how our behaviour modelling approach, introduced in previous chapters, has been extended with *visual generative design patterns* will be discussed. This allows creating generative design patterns and including them into our existing graphical behaviour modelling language.

The design patterns known from Software Engineering are usually descriptive. They are described in terms of their participants and their collaborations. Furthermore, the descriptions also often include implementation issues and sample code. This information can be used by programmers to use and implement such a design pattern. However, a textual description is not sufficient for describing generative design patterns and not appropriate for *visual* generative design patterns. Therefore, a behavioural design patterns framework that allows combining our overall behaviour modelling approach with visual generative design patterns has been developed.

When developing the framework, a number of requirements had to be taken into account. Firstly, it must be possible to manage the collection of behavioural patterns and to easily construct new patterns and add them to the existing collection. Secondly, the usage of the behavioural patterns available in the framework needs to be supported as much as possible. Since non-VR-experts are part of the target public, these users should be guided in the process of using (instantiating) a pattern in the design of a particular Virtual Environment through the graphical modelling language. Thirdly, it must be possible to adapt a generative design pattern to a particular context, e.g., it must be possible to do small adjustments and to specify parameters. A mechanism must be provided to facilitate these adaptations.

Figure 6.1 gives a detailed overview of the complete framework. In the framework, basically two major flows can be detected, that is the ”Pattern Creation” flow and the ”Pattern Usage” flow. Recalling figure 4.1 in chapter 4 (on page 80) about the extended architecture, allows us to allocate these flows to the different levels of the behaviour modelling approach.

- **Pattern Creation.** The pattern creation flow represents the process of creating a new behavioural design pattern and making it available in our graphical notation. It consists of three consecutive steps namely the *Pattern Specification*, the *Extension Specification* and the *Source*

Code Specification. This process fits into the Meta Level of our modelling approach since it provides new modelling constructs to the designer that can then be used for specifying the behaviour of a Virtual Environment.

- **Pattern Usage**. The pattern usage represents the process of using a pattern in the behaviour specifications and how to adapt it to a particular context (for a particular Virtual Environment). It also involves three steps namely the *Selection*, the *Adaptation* and the *Generation*. The usage process can be located in the behaviour specification at the Domain Level of our modelling approach. Using a pattern and applying it to a particular Virtual Environment is similar to creating a behaviour definition and a behaviour invocation respectively.

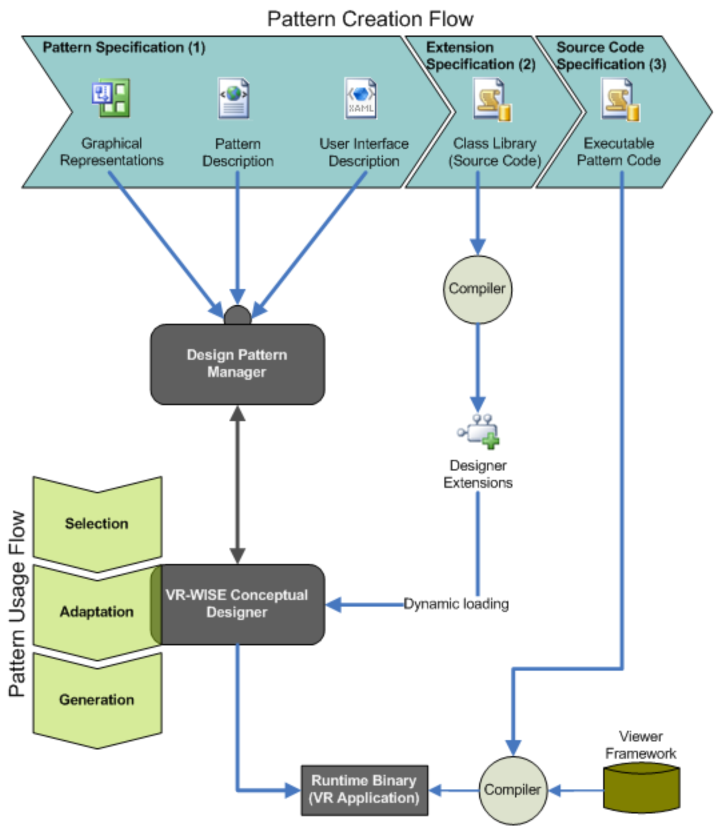

Pattern Creation Flow
Pattern Specification (1)
Graphical Representations
Pattern Description
User Interface Description
Extension Specification (2)
Class Library (Source Code)
Source Code Specification (3)
Executable Pattern Code
Compiler
Design Pattern Manager
Designer Extensions
Pattern Usage Flow
Selection
Adaptation
Generation
VR-WISE Conceptual Designer
Dynamic loading
Viewer Framework
Runtime Binary (VR Application)
Compiler

Figure 6.1: *Behavioural design pattern framework*

At the top of figure 6.1, the pattern creation flow is represented (by the large horizontal arrow). It denotes the different steps that need to be followed in order to construct a new pattern in our framework. This process is further explained in section 6.3.1. The specifications that result from creating a pattern are processed by a tool called *Design Pattern Manager*. The Design Pattern Manager is a separate application which is added to our toolbox and can be used to interpret the specifications, process them and make them available for its use in the *VR-WISE Conceptual Designer* application. Furthermore, the Design Pattern Manager allows us to properly manage all the behavioural design patterns in our framework (organize them, create new ones, and so on). The VR-WISE Conceptual Designer is the software tool supporting the conceptual design phase of the VR-WISE approach. It is a diagram editor that allows creating the conceptual models in a graphical way. This tool is extended to be able to facilitate the use of the patterns in our graphical modelling language. From the different conceptual models that are created, representing the static part as well as the dynamic part of the Virtual Environment (possibly using the behavioural design patterns), the VR-WISE Conceptual Designer can then generate code. The code that is generated is a specification of the static scene combined with the source code of the behaviours to be used as input to initialize the runtime binary (see figure 6.1), i.e. the actual Virtual Reality application. Note that both the code of the static scene and the code for the behaviours are application specific. This process is further explained in section 6.3.2. More details about the VR-WISE Conceptual Designer itself can be found in chapter 7.

6.3.1 Building Patterns

The creation of a new behavioural pattern is not an easy task and it requires more experienced designers having some programming skills. In other words, this step is not intended for non-VR-experts. The creation of new patterns requires designers who are more experienced in Virtual Reality since, for some parts of the specifications, notions about technologies related to Virtual Reality are needed. It involves specifying a number of things. There are three different sequential steps in the creation process of a behavioural pattern, namely the *pattern specification*, the *extension specification* and the *source code specification* (see top of figure 6.1). The majority of the work needs to be done in the pattern specification.

The pattern specification (1) allows the designer to describe, at a high level, the front-end for the behavioural pattern. This is the part that will be seen by the user of the pattern. The extension specification (2) allows the pattern designer to describe the back-end of the behavioural pattern. This part is not seen by the user of the pattern, but is required in order to be able to generate programming code. The source code specification (3) allows the pattern designer to provide the executable pattern code, which is the actual code that implements the behavioural pattern.

6.3.1.1 Graphical Elements Specification (Pattern Specification)

The first specification that needs to be created is a so-called stencil. A stencil is a container of all the graphical elements necessary for constructing the particular pattern. These elements are the graphical representations of the modelling concepts needed in the pattern. A number of predefined graphical elements are already available to the pattern designer (see figure 6.2).

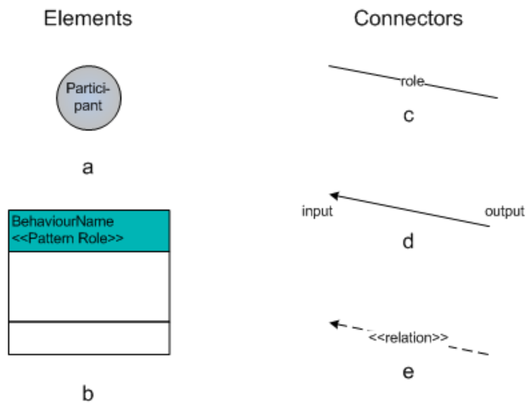


Figure 6.2: *Default pattern creation elements*

The *actor* (6.2a), denoted by a circle, represents an object (participant) that is involved in a pattern. The name of the participant is specified inside the element. The *behavioural* element (6.2b) usually represents a behavioural aspect of the pattern, that is the algorithm behind the pattern. The name as well as the role this element has within the pattern is noted in the top area of the element. The middle area can be used to hold custom information or even a sub-diagram while the bottom area usually contains some textual information (such as a script or any particular attributes). The *participant link* (6.2c) is used to connect a participant to a behavioural element and holds the name of the role of this participant within the pattern. The *data link* (6.2d) is used to connect the output of one behavioural element with the input of another behavioural element. The *causality link* (6.2e) is used to connect two behavioural elements and denotes a causal relation between the two elements. Textual information referring to the relationship between the two elements can be noted on the link itself.

With these predefined graphical elements, the objective is to avoid that different pattern designers start creating their own graphical elements, which would result in large and inconsistent sets of graphical elements. After a while, this would most probably lead to an explosion in terms of the number of elements, which then results in the fact that these graphical elements are more difficult to use and to remember by other users. Because of the existence of these predefined graphical elements, it may be unnecessary to create a new stencil for some patterns. To facilitate a seamless integration of the behavioural patterns in our general behaviour modelling approach, it was decided to use graphical elements that are the same or close to the ones that were already introduced, as seen earlier in the dissertation. If the default graphical elements provided by the framework are sufficient for the pattern, no new stencil needs to be created. However, the framework allows a pattern designer to add new ones, as it is impossible to provide all the different graphical elements for all the different patterns that might be constructed in the future. Therefore, the pattern designer may create his own graphical elements in addition to the ones already provided. In this way, the framework is extendable. The only thing the designer has to adhere to, is to give the graphical element a proper type of shape (*ShapeType*) so that it can be referred to later on.

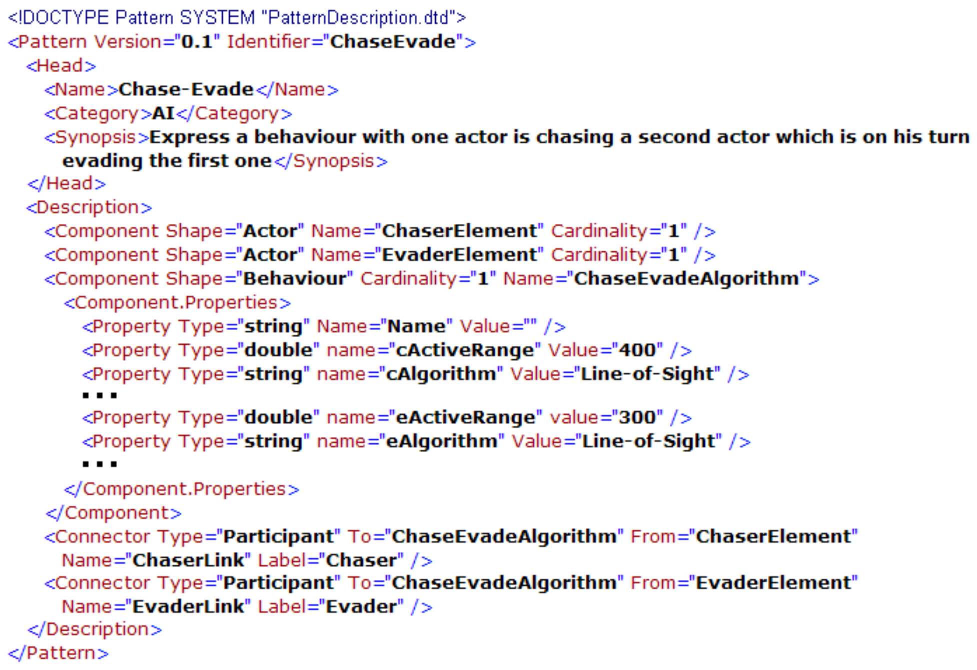
6.3.1.2 Pattern Description (Pattern Specification)

The second specification is the actual pattern description. Once all the different graphical elements have been identified (and possibly new graphical elements have been added), the pattern designer needs to specify how the pattern is composed using the graphical elements from the stencil. He must specify how many instances of a particular element are needed, which roles they play, how the elements are connected to each other, and which parameters are required. Also default values must be provided for all parameters. This is needed to provide as much support as possible to the designers. This specification is captured in a standard XML format, which has the benefit of being tool independent. External tools can be used for building this specification and it can also be easily validated against a DTD.

Let us first start with a small example. Later, more formal definitions are given about how such a specification should be constructed. Figure 6.3 gives an extract of a pattern description. In the pattern that is described here, only the standard graphical elements are used. The pattern called Chase-Evade contains exactly three elements, two *actor* elements and a single *behavioural* element. The actors (participants) involved in the pattern play the role of the chaser, respectively the evader. This is indicated by the two connectors, namely two participant links connecting the actors with the behavioural element. The behavioural element represents the actual behaviour (the algorithm behind the pattern) in which the actors take part. A detailed description of the pattern is omitted here but will be discussed in section 6.4.

Figure 6.4 shows how the pattern is visually represented in our framework based on the specification given in figure 6.3. One can clearly see the correspondences between the specification and the actual visual pattern. Also note the correspondence with the graphical elements discussed earlier.

Each pattern description is specified by means of a single XML docu-



```
<!DOCTYPE Pattern SYSTEM "PatternDescription.dtd">
<Pattern Version="0.1" Identifier="ChaseEvade">
  <Head>
    <Name>Chase-Evade</Name>
    <Category>AI</Category>
    <Synopsis>Express a behaviour with one actor is chasing a second actor which is on his turn
      evading the first one</Synopsis>
  </Head>
  <Description>
    <Component Shape="Actor" Name="ChaserElement" Cardinality="1" />
    <Component Shape="Actor" Name="EvaderElement" Cardinality="1" />
    <Component Shape="Behaviour" Cardinality="1" Name="ChaseEvadeAlgorithm">
      <Component.Properties>
        <Property Type="string" Name="Name" Value="" />
        <Property Type="double" name="cActiveRange" Value="400" />
        <Property Type="string" name="cAlgorithm" Value="Line-of-Sight" />
        ...
        <Property Type="double" name="eActiveRange" value="300" />
        <Property Type="string" name="eAlgorithm" Value="Line-of-Sight" />
        ...
      </Component.Properties>
    </Component>
    <Connector Type="Participant" To="ChaseEvadeAlgorithm" From="ChaserElement"
      Name="ChaserLink" Label="Chaser" />
    <Connector Type="Participant" To="ChaseEvadeAlgorithm" From="EvaderElement"
      Name="EvaderLink" Label="Evader" />
  </Description>
</Pattern>
```

Figure 6.3: *Pattern description of the Chase-Evade pattern*

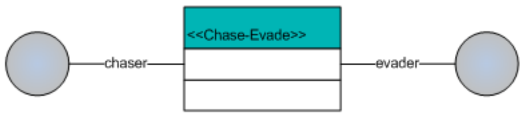


Figure 6.4: *Visual representation of the Chase-Evade pattern*

ment. To specify a pattern, a particular format needs to be followed as defined by the declarations below.

```
<!ELEMENT Pattern (Head, Description)>
<!ATTLIST Pattern
  Identifier ID #REQUIRED
  Version CDATA #IMPLIED
>
```

The document begins with a root element called *Pattern*. This element has a required attribute *Identifier* and one optional attribute *Version*. The purpose of the Identifier is to uniquely identify the pattern. The Version allows the designer to give some information about this particular version of the pattern. The Pattern element furthermore consists of two parts, i.e. the *Head* and the *Description*.

The first part of the pattern description is the *Head* which is used to specify any meta-information about the pattern.

```
<!ELEMENT Head ((Name)+, (Category)*, (Synopsis)*)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Category (#PCDATA)>
<!ELEMENT Synopsis (#PCDATA)>
```

The Head element contains a sequence of child elements. There can be one or more *Name* elements defined for a single pattern since a pattern can have, besides its name, also a number of aliases. There can be zero or more *Category* elements. These elements enable a better classification of the pattern, which facilitates its use. Finally, there can be zero or more *Synopsis* elements giving a more detailed description of the pattern. All child elements can contain any kind of data.

The second part of the pattern description is the *Description* which is used to define the actual pattern itself. As mentioned before, a pattern is built from a number of graphical elements connected in a certain way. This information is specified here.

```
<!ELEMENT Description ((Component | Connector)*)>

<!ELEMENT Component ((Description)?, (Component.Properties)?)>
<!ATTLIST Component
  Name ID #REQUIRED
  Shape NMTOKEN #REQUIRED
  Cardinality CDATA #REQUIRED
>

<!ELEMENT Connector ((Connector.Properties)?)>
<!ATTLIST Connector
  Name ID #REQUIRED
  Shape NMTOKEN #REQUIRED
  From IDREF #REQUIRED
  To IDREF #REQUIRED
  Label CDATA #IMPLIED
  FromLabel CDATA #IMPLIED
  ToLabel CDATA #IMPLIED
>
```

The Description element is having zero or more child elements. These elements can be either *Component* elements or *Connector* elements which can be given in any order.

A Component element represents a graphical element. It has two required attributes namely a unique *Name* by which this element can be referred in the pattern description and a *Shape* which refers to the name of the shape that is used from the default shapes or that is created by the designer. In other words, this attribute provides the link between the pattern description and the graphical elements specification (previous step). The third attribute is the *Cardinality* which allows us to put constraints on the number of elements used, i.e. how many elements of this particular kind can be used in the pattern description. Possible values for this attribute are n (a single value) or n-m (a lower bound and upper bound). Furthermore, a Component can have zero or one Description elements as child element. This allows the designer to create more complex patterns (e.g., patterns in which graphical elements have sub-elements).

The Connector element represents a link between two graphical elements (or in other words, between two Component elements). As with the Component, it has a unique *Name* attribute and a *Shape* attribute. It furthermore has a source as given by the *From* attribute and a target as given by the *To* attribute. The connector points from the element referenced in the From attribute to the element referenced in the To attribute. These attributes hold the unique names of the Component elements. Next, it has three more attributes namely the *Label*, *FromLabel* and *ToLabel* which hold the information that is written on the connectors in the graphical notation. Depending on which kind of connector, some are used and others are omitted.

```
<!ELEMENT Element.Properties (Property*)>
<!ELEMENT Connector.Properties (Property*)>

<!ELEMENT Property EMPTY>
<!ATTLIST Property
  Name ID #REQUIRED
  Label CDATA #IMPLIED
  Value CDATA #IMPLIED
  Type (bool | string | double | integer | array) #REQUIRED
>
```

Finally, the Component elements can have a property block called *Element.Properties*. Also the Connector elements can have a property block which is called *Connector.Properties*. These two blocks allow specifying the properties for the elements and are defined in the same way. They can contain zero or more *Property* elements. A Property element can be defined as having a unique *Name* to refer to it later on, and an optional *Label* giving a more intuitive name for it. Furthermore, it also has a *Type* representing the type that the value of the property must adhere to and a *Value* part which is the default value for this property.

6.3.1.3 User Interface Description (Pattern Specification)

The third specification is the user interface description. As mentioned above, the user of a pattern should be able to change the default values for parameters defined for the different elements composing the pattern. To allow this, each element of the pattern can be associated with a custom dialog, which needs to be specified by the pattern designer. This dialog is described by means of a user interface specification language. In our case, Extensible Application Markup Language (XAML) is used [Nathan, 2006]. XAML is used for two reasons. Firstly, in the future, it can be automatically incorporated in the framework without having to be parsed and interpreted explicitly. Secondly, it has a powerful data-binding mechanism that can be used for free. The specification contains the different GUI controls that an end-user can use to enter the values for the parameters. For example, a textbox for a string parameter, a checkbox for a boolean parameter, a listbox for an array parameter. Please note that not the complete XAML specification language is supported at this moment. Only the most frequently used GUI controls can be interpreted by our framework. Furthermore, a data-binding mechanism provides a way to create a 'connection' between the controls in the dialog and the actual data in the pattern descriptions mentioned earlier. The controls are automatically updated when the data in the pattern changes and vice versa changes in the controls are automatically propagated in the underlying pattern instantiation. For example, a particular textbox can be bound to a string parameter of a pattern element. If the user edits the value in the textbox element, the underlying data value is automatically updated to reflect that change.

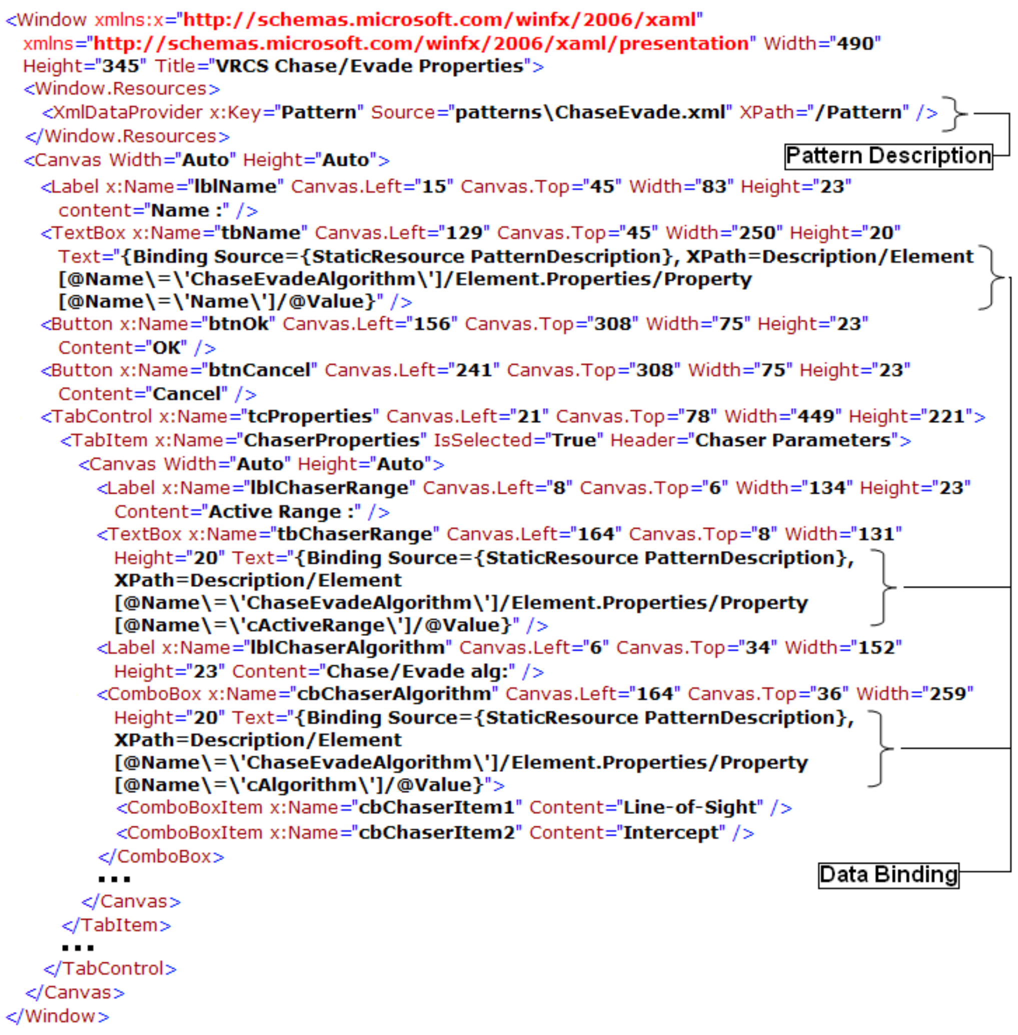
Let us also start with reconsidering the small example mentioned earlier. Later on, more formal definitions about the different constructs available in our framework are briefly discussed. Figure 6.5 gives an extract of a user interface description. One can clearly see the link that is established between the user interface description and the pattern description (i.e. patterns/ChaseEvade.xml) that was created before. We have marked this in the figure. Furthermore, also the different data-bindings between the properties of the pattern and the controls in the dialog are clearly visible as marked in the figure as well.

Figure 6.6 shows the dialog that is generated by our framework based on the specification given in figure 6.5. One can see the correspondences between the specification and the actual dialog. Also note the default values that were defined in the pattern description discussed earlier.

A user interface description is a XAML document that is specified according to a fixed format. This format is given by the following declarations.

```
<!ELEMENT Window (Window.Resources, Canvas)>
<!ATTLIST Window
  Title CDATA #REQUIRED
  Height CDATA #REQUIRED
  Width CDATA #REQUIRED
>
```

The document begins with a root element called *Window* denoting that a dialog window is being defined. The Window element requires a number of attributes to be given. A *Title* is needed which is represented in the



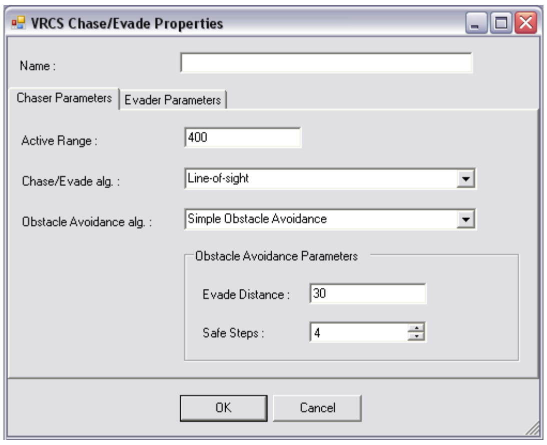
```
<Window xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
  xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation" Width="490"
  Height="345" Title="VRCS Chase/Evade Properties">
  <Window.Resources>
    <XmlDataProvider x:Key="Pattern" Source="patterns\ChaseEvade.xml" XPath="/Pattern" />
  </Window.Resources>
  <Canvas Width="Auto" Height="Auto">
    <Label x:Name="lblName" Canvas.Left="15" Canvas.Top="45" Width="83" Height="23"
      content="Name :" />
    <TextBox x:Name="tbName" Canvas.Left="129" Canvas.Top="45" Width="250" Height="20"
      Text="{Binding Source={StaticResource PatternDescription}, XPath=Description/Element
      [@Name\=\'ChaseEvadeAlgorithm\']/Element.Properties/Property
      [@Name\=\'Name\']/@Value}" />
    <Button x:Name="btnOk" Canvas.Left="156" Canvas.Top="308" Width="75" Height="23"
      Content="OK" />
    <Button x:Name="btnCancel" Canvas.Left="241" Canvas.Top="308" Width="75" Height="23"
      Content="Cancel" />
    <TabControl x:Name="tcProperties" Canvas.Left="21" Canvas.Top="78" Width="449" Height="221">
      <TabItem x:Name="ChaserProperties" IsSelected="True" Header="Chaser Parameters">
        <Canvas Width="Auto" Height="Auto">
          <Label x:Name="lblChaserRange" Canvas.Left="8" Canvas.Top="6" Width="134" Height="23"
            Content="Active Range :" />
          <TextBox x:Name="tbChaserRange" Canvas.Left="164" Canvas.Top="8" Width="131"
            Height="20" Text="{Binding Source={StaticResource PatternDescription},
            XPath=Description/Element
            [@Name\=\'ChaseEvadeAlgorithm\']/Element.Properties/Property
            [@Name\=\'cActiveRange\']/@Value}" />
          <Label x:Name="lblChaserAlgorithm" Canvas.Left="6" Canvas.Top="34" Width="152"
            Height="23" Content="Chase/Evade alg:" />
          <ComboBox x:Name="cbChaserAlgorithm" Canvas.Left="164" Canvas.Top="36" Width="259"
            Height="20" Text="{Binding Source={StaticResource PatternDescription},
            XPath=Description/Element
            [@Name\=\'ChaseEvadeAlgorithm\']/Element.Properties/Property
            [@Name\=\'cAlgorithm\']/@Value}">
            <ComboBoxItem x:Name="cbChaserItem1" Content="Line-of-Sight" />
            <ComboBoxItem x:Name="cbChaserItem2" Content="Intercept" />
          </ComboBox>
          ...
        </Canvas>
      </TabItem>
      ...
    </TabControl>
  </Canvas>
</Window>
```

Figure 6.5: *User interface description of Chase-Evade pattern*

border of the dialog window. The *Height* and *Width* attributes determine the size of the window. Next to the attributes, the Window element also contains two parts, i.e. a *Window.Resources* child element and a *Canvas* child element.

The first part of the user interface description is the *Window.Resources* element which is used to specify the resources (data) that are used inside this specification.

```
<!ELEMENT Window.Resources ((XmlDataProvider))>
<!ELEMENT XmlDataProvider EMPTY>
<!ATTLIST XmlDataProvider
  Source CDATA #REQUIRED
  XPath CDATA #FIXED "Pattern/Description"
```


VRCS Chase/Evade Properties

Name :

Chaser Parameters | Evader Parameters

Active Range : 400

Chase/Evade alg. : Line-of-sight

Obstacle Avoidance alg. : Simple Obstacle Avoidance

Obstacle Avoidance Parameters

Evade Distance : 30

Safe Steps : 4

Figure 6.6: *Dialog of the Chase-Evade pattern*

```
  x:Key CDATA #FIXED "PatternData"
>
```

The Window.Resources element contains one child element namely the *XmlDataProvider* that is made available in XAML to provide easy access to XML documents. An XmlDataProvider has three attributes. Firstly, the *Source* is used to specify a URI which refers to a local file, or a file on the Internet that holds the pattern description. Secondly, the *XPath* attribute is set to an XPath[1] query that tells where the data is to be found in the XML document. In our case, this has a fixed value since, as was seen earlier, our pattern descriptions have a fixed structure. Thirdly, the *Key* attribute allows giving a name for this resource which can be used later on in the specification to do the data-bindings.

The second part of the user interface description is the *Canvas* which is used to define the different GUI controls of the dialog window. As mentioned before, a dialog is built to contain a number of controls for the data in the pattern. This information is specified here.

```
<!ELEMENT Canvas ((TextBox | Label | Button | ComboBox
                   ListBox | CheckBox | TabControl)*)>
<!ATTLIST Canvas
  Height CDATA #IMPLIED
  Width CDATA #IMPLIED
>

<!ENTITY % attributes "x:Name ID #REQUIRED
  Canvas.Left CDATA #REQUIRED
  Canvas.Top CDATA #REQUIRED
  Height CDATA #REQUIRED
```

[1]XML Path Language http://www.w3.org/TR/xpath

```
  Width CDATA #REQUIRED"
>
```

The Canvas is a very basic panel only supporting the notion of positioning elements with explicit coordinates. A *Height* and *Width* attribute can also be given for the Canvas element but they are optional. A Canvas can contain zero or more controls. The controls that are supported are *TextBox*, *Label*, *Button*, *ComboBox*, *ListBox*, *CheckBox* and *TabControl*. These controls are defined by a unique name as given by the *Name* attribute. Furthermore, a *Canvas.Left* and *Canvas.Top* attribute should be given to locate them on the canvas as well as a *Height* and *Width* attribute defining their size.

In addition, the controls might have a *Text* or *Content* attribute which can be bound to a property in the pattern description (not given in the declaration). This is done using an XPath expression returning the correct data.

Please note that it might have been possible to use generalized interfaces for the parameter dialogs, one that could suit for all kinds of patterns (pattern elements). Although this step seems to require additional overhead of specifying a parameter dialog for each pattern, we choose to still include it in our process. This allows the dialogs to be created specifically for the pattern (or pattern element) at hand which would result in a more user-friendly interface than if it would have been generated automatically.

6.3.1.4 Class Library (Extension Specification)

In the extension specification, a collection of classes (a class library) written in a programming language (C# in our case) is created. It describes the actual semantics of the visual pattern. This specification is a necessary step in order for the framework to know how to process an instantiated pattern so that code related to that pattern can be automatically generated. The purpose of this code is to enable the correct interpretation of the newly specified pattern together with all its parameters and load it into our internal data structure. So, it is capable of reading all the possible variations of the pattern that can be specified. For example, if multiple graphical elements of an actor (participants) are allowed, it should be able to cope with multiple graphical elements of the actor (participants). It also describes what code should be generated once a pattern has been instantiated. This class library is compiled to become a so-called *designer extension* (see figure 6.1). This extension is in fact a small component that can be dynamically loaded into our VR-WISE Conceptual Designer.

Let us illustrate this step for our small example. A number of classes need to be created. At least one class is required to have the functionality to be able to parse the graphical pattern correctly. In our example, we know that when a pattern is instantiated we can only have one behavioural element and exactly two actor elements (chaser and evader) as given by the cardinality constraints in the pattern description. Next, the properties need to be dealt with. Here, only the behavioural element has some properties defined for it. These properties need to be extracted and stored for later use. Another class is needed to be able to generate the correct source code based on the information that was extracted from the instantiated pattern. This source code should actually instantiate the classes that will be created in the next step.

6.3.1.5 Executable Pattern Code (Source Code Specification)

Until now, everything to instantiate the pattern, to process it and to generate initialization code for it, is specified. However, the actual implementation of the behaviour represented by the pattern is not yet specified. This is done in the executable pattern code step. This specification is also a collection of classes; the classes necessary to execute the pattern or the algorithm that one wants to use. There are two possibilities. Either, the pattern designer creates his own implementation in which case programming skills are required. Or, the pattern designer can reuse an already existing implementation or algorithm and incorporate it completely as a black box. In the latter case, usually some wrapper classes need to be written to have the external classes working properly together with the rest of the framework classes. The classes will be compiled together with our viewer framework to come to a runtime binary (the resulting VR application). These are the classes that should be used (or instantiated) in the code generated by the VR-WISE Conceptual Designer.

For implementing the behavioural pattern in our small example, a total of seven classes are needed. A first class needs to be implemented to provide the actual chase-evade behaviour. The pattern also requires a few helper algorithms to be specified. These need to be implemented too. An abstract class and two concrete classes are used for implementing two different types of chase-evade algorithms that may be used in this pattern (either the "line-of-sight" chasing or the "intercept" chasing algorithm). Also an abstract class and two concrete classes are used for implementing the obstacle avoidance algorithms available for this pattern (either a "simple" or a "target-based" obstacle avoidance algorithm). More details on the meaning of the different algorithms are given in the discussion on the Chase-Evade pattern in section 6.4.

6.3.2 Using Patterns

Once a behavioural pattern has been created, it is interpreted by our Design Pattern Manager and thereby made available to the designer. It can then be used (called instantiated) and applied in a particular Virtual Environment by both designers experienced in Virtual Reality and those who are less experienced in Virtual Reality. The process of instantiating a behavioural pattern consists of three steps.

6.3.2.1 Selection

In the first step, the selection, the designer selects an appropriate pattern from the collection of patterns available in the framework (and created earlier by a pattern designer). The different graphical elements necessary for this particular pattern are automatically dropped on the drawing canvas of the VR-WISE Conceptual Designer and the proper connections are made. It is important that the pattern collection is organized well in order to facilitate the selection of a pattern as much as possible.

6.3.2.2 Adaptation

In the second step, the adaptation, the designer adapts the pattern to the particular context of the application. Two different kinds of pattern adaptation are supported. The first kind of adaptation is by giving proper values to the parameters (overriding the default values provided by the pattern designer). This can be done through the dialogs that were specified in the user interface description. The second kind of adaptation is by adding extra graphical elements and/or removing some of the graphical elements that were automatically dropped on the drawing canvas. The graphical elements that can be added are the ones available in the stencil of the pattern. Of course, there are some limitations; the instantiation of the pattern needs to respect the constraints associated with the pattern (i.e. cardinalities on the graphical elements) to avoid ending up with something that has nothing to do with the pattern anymore. These constraints are defined in the pattern description as mentioned earlier.

6.3.2.3 Generation

The third step, the generation, is the automatic generation in which all the specifications are translated into the correct application specific initialization code that serves as the input for the resulting Virtual Reality application. The code that is generated by the VR-WISE Conceptual Designer consists of two parts: firstly a specification of the static scene, and secondly the source code of the behaviours (instantiated behavioural patterns). Both parts are to be used as input to initialize the runtime binary that was mentioned earlier. Note that only the code of the static scene and the code for the behaviours are specific for the particular application, the runtime binary itself is not.

The process of using a behavioural design pattern will be illustrated by means of an example in the following section.

6.4 A Collection of Behavioural Design Patterns

In previous sections, our behavioural design pattern framework was discussed. The creation flow and the usage flow were explained and illustrated by means of an example. This framework facilitates the use of visual generative design patterns in the behaviour specifications. This approach allows to easily incorporate existing VR expertise. Complex behaviours can now be defined more quickly and with fewer errors by making use of existing behavioural patterns. Also the size and the complexity of the diagrams can be reduced by using the pattern-based framework. This results in models that are easier to read and better maintainable. Furthermore, since the design patterns encapsulate a larger piece of functionality and fixed collaborations between the actors (participants), optimizations and existing algorithms can be used for the code generation, which will result in code that is more efficient than if it would have been generated directly from conceptual models, i.e. by only using the standard elements of our graphical modelling language.

This section reviews some of the most frequently used patterns that have been encountered so far in the different Virtual Environments built using our VR-WISE design approach. All these patterns were successfully created within our pattern-based framework. It also shows the feasibility of our framework.

As a way to illustrate the different patterns, an example Virtual Reality application is provided. In previous chapters, the example of a virtual department store has been used regularly. In the following, a larger example is used and a complete virtual city application is being developed, which has been created using the VR-WISE Conceptual Designer module. The Virtual Environment has a virtual city park that contains monuments, a road network with cars and buses driving around, and a number of buildings, which can be visited.

In the remaining part of this section, a number of patterns, which have been created using our approach, will be explained. Each pattern will be described using the following structure:

- **Name.** First, the name of the pattern is given. This is an important item since it allows referring to the pattern in a discussion between the different stakeholders of the Virtual Environment.

- **Description.** The actual description of the pattern starts with a concise explanation of the general intent of the pattern. It answers the question on what the pattern does, or what problem it tries to solve.

- **Motivation.** Some examples of the pattern in existing Virtual Environments are given to illustrate the design problem. This will help to understand the more abstract description of the pattern.
- **Structure.** Then, an explanation of the different elements that compose the pattern is given. It discusses the actors and/or behaviours participating in the design pattern, how they are graphically connected to each other and what their responsibilities are. This section also discusses the different parameters that are required (and need to be entered by the pattern user).
- **Usage.** Afterwards, a more detailed example (in the context of the virtual city application) is given to illustrate how the pattern is to be used.
- **[Sample Code].** Finally, some code examples are included showing what is actually generated by the framework and what is not. This section is optional, not all patterns will contain full descriptions of the generated source code.

Please note that this collection of patterns is not exhaustive. It is continuously being extended with new patterns.

6.4.1 Chase-Evade Pattern

6.4.1.1 Name

Chase-Evade

6.4.1.2 Description

The goal for this pattern is to express the behaviour where a so-called ”chaser” tries to capture (or chase) another actor called the ”evader”. The evader on his turn tries to evade (or avoid being captured by) the chaser.

6.4.1.3 Motivation

The chase-evade pattern is often encountered in Virtual Environments or Game Worlds involving some kind of Artificial Intelligence. Examples can be found in action games where a guard has detected an intruder on the premises and tries to take him into custody, or in role-playing games where an enemy either tries to capture the player or run away from it. In the context of our virtual city application, the chase-evade pattern was used to allow an avatar (e.g., a tourist in the Virtual Environment) to smoothly intercept the driving bus and once it gets close enough, it can then get on board of the bus and take the tour. Another example is a police officer trying to catch a thief of the department store in the city.

6.4.1.4 Structure

A chase-evade pattern is defined using three elements, one behavioural element and two actor elements (see figure 6.7). The actors (participants) involved in the pattern play the role of the chaser respectively the evader. This is indicated by participant links connecting the actors with the behavioural element. The behavioural element represents the actual behaviour (chase-evade algorithm) in which the actors take part.

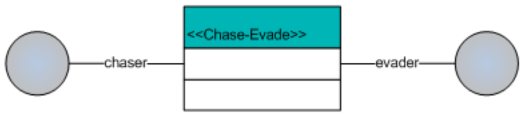


Figure 6.7: *Uninitialized Behaviour Definition Diagram of the Chase-Evade pattern*

Figure 6.7 shows how the pattern is graphically represented in our framework. As you might have noticed, the pattern described here is similar to the one that was described in the previous section.

The pattern requires a number of parameters to be given for the chaser-participant as well as for the evader-participant. The *active range* parameter is used to specify when the chase-evade has to be initiated (given for both the chaser and the evader). The second parameter is needed to indicate the actual algorithm that should be used for performing the chase and evade. This algorithm can be the simple *line-of-sight* chasing, which just corrects the position of the chaser based on the position of the evader. In this way, their distance is reduced (and the opposite for the evader). But the algorithm can also be the *intercept* chasing where the chaser takes the heading of the evader into account to try to intercept it in a smarter way. The third group of parameters is to indicate the obstacle avoidance algorithm that needs to be used. Here, there are again two possibilities, a *simple* one, where a random direction is chosen when avoiding an obstacle, and a *target-based* version, where the direction in which the actor moves around an obstacle depends on a target.

6.4.1.5 Usage

Figure 6.8 shows a Behaviour Definition Diagram illustrating how the pattern is used. The Police Officer and Thief actors respectively play the roles of chaser and evader. The behavioural element is called ChaseThief.

Figure 6.9 shows the parameter dialog for the chase-evade pattern that is invoked by double-clicking the behavioural element. This dialog can be used to override the default values for the parameters.

Remember that in order to use this behaviour in the context of our particular virtual city application, the actors need to be linked with actual

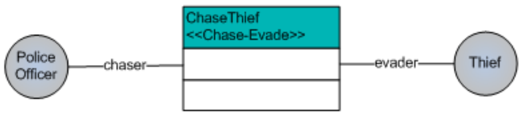
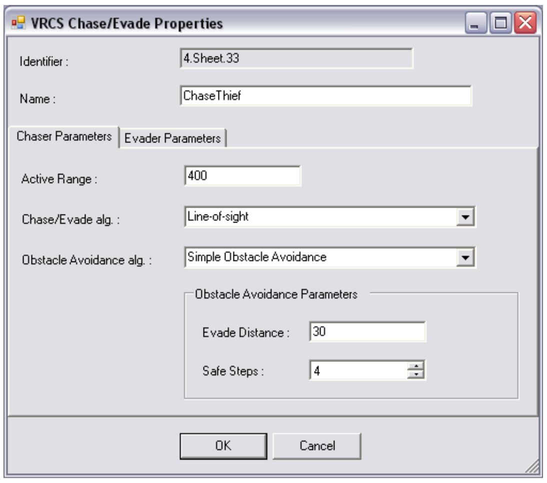


Figure 6.8: *Behaviour Definition Diagram using the Chase-Evade pattern*



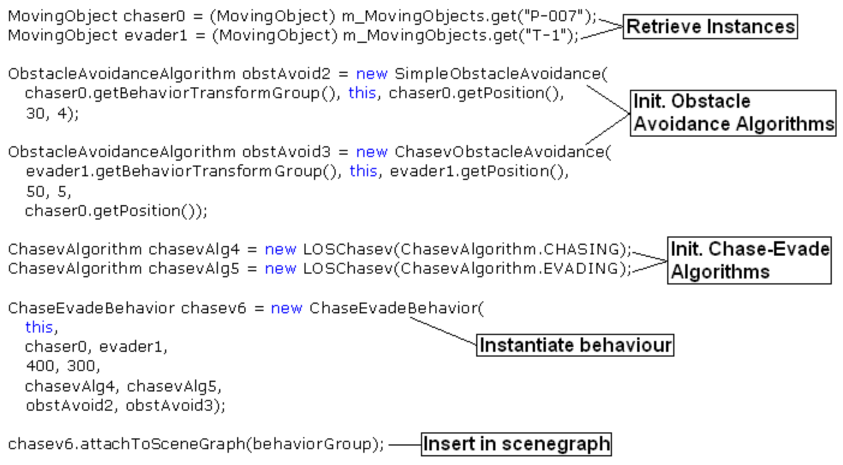
Identifier : 4.Sheet.33
Name : ChaseThief
Active Range : 400
Chase/Evade alg. : Line-of-sight
Obstacle Avoidance alg. : Simple Obstacle Avoidance
Obstacle Avoidance Parameters
Evade Distance : 30
Safe Steps : 4

Figure 6.9: *Parameter dialog for the Chase Evade pattern*

objects from the Virtual Environment by means of a Behaviour Invocation Diagram. Suppose that in our example, the actors Police Officer and Thief, are linked to respectively the ”P-007” and ”T-1” instances. The Behaviour Invocation Diagram for this is not given here.

6.4.1.6 Example (of generated) Code

The piece of code in figure 6.10 gives a small extraction of the pattern initialization code. Please remember that it is not the complete algorithm generated when the pattern is used. It is only the source code necessary for invoking (initializing) the algorithm that is shown. The actual algorithm is specified by the pattern designer and is shielded from the behaviour designer. In the extraction, the first two lines show the instances (”P-007” and ”T-1”) that are corresponding to the actors being retrieved. Then, the obstacle avoidance algorithms are initialized, first for the chaser, then for the evader. Afterwards, the chase-evade algorithms are initialized with the correct values for the parameters entered through the dialog which is firstly the algorithm for the chasing-part and secondly the algorithm for the evading-part. Then, the actual pattern behaviour is initialized with all this information. Finally, this behaviour is attached to the scene graph (i.e. behaviorGroup), which is necessary for it to become active in the Virtual Reality application.



```
MovingObject chaser0 = (MovingObject) m_MovingObjects.get("P-007");
MovingObject evader1 = (MovingObject) m_MovingObjects.get("T-1");

ObstacleAvoidanceAlgorithm obstAvoid2 = new SimpleObstacleAvoidance(
  chaser0.getBehaviorTransformGroup(), this, chaser0.getPosition(),
  30, 4);

ObstacleAvoidanceAlgorithm obstAvoid3 = new ChasevObstacleAvoidance(
  evader1.getBehaviorTransformGroup(), this, evader1.getPosition(),
  50, 5,
  chaser0.getPosition());

ChasevAlgorithm chasevAlg4 = new LOSChasev(ChasevAlgorithm.CHASING);
ChasevAlgorithm chasevAlg5 = new LOSChasev(ChasevAlgorithm.EVADING);

ChaseEvadeBehavior chasev6 = new ChaseEvadeBehavior(
  this,
  chaser0, evader1,
  400, 300,
  chasevAlg4, chasevAlg5,
  obstAvoid2, obstAvoid3);

chasev6.attachToSceneGraph(behaviorGroup);
```

Retrieve Instances
Init. Obstacle Avoidance Algorithms
Init. Chase-Evade Algorithms
Instantiate behaviour
Insert in scenegraph

Figure 6.10: *Code extraction*

6.4.2 Pattern Movement Pattern

6.4.2.1 Name

Pattern Movement

6.4.2.2 Description

This pattern allows controlling the movement of an object in the Virtual Environment according to some predefined path specified by a series of key locations along the trail. It is a way to give the illusion of intelligent behaviour since the movement makes the object appear as if it is executing complex manoeuvres.

6.4.2.3 Motivation

Within dynamic Virtual Environments, objects often move along a predefined path. For example, in our virtual city application, buses need to drive along the streets according to a predefined route and stop at different places. This sort of behaviour occurs in many Virtual Environments and games, e.g., a guided tour where tourists are following a predefined path, in First-Person-Shooter (fps) games where a guard is patrolling around a castle in a well-defined manner, or in flight simulator games where evasive manoeuvers could be taken according to a fixed pattern.

6.4.2.4 Structure

The basic structure of a pattern movement pattern consists of an actor element and a behavioural element connected to each other by means of a

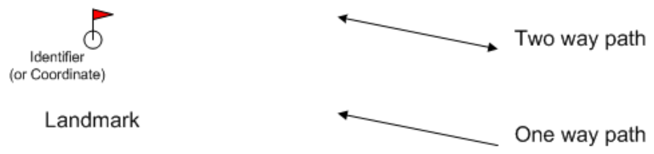


Figure 6.11: *Additional graphical elements for the Pattern Movement pattern*

participant link. Furthermore, the actual path is defined by means of a graph consisting of minimal two *landmarks* (indicated by a small flag), and *path specifiers* (represented by a one headed or a two headed arrow) connecting the landmarks. The landmark and path specifier are two new graphical elements that have been defined in the stencil of this pattern (see figure 6.11a and 6.11b for their graphical representation). The landmarks represent the key locations of the movement. They are specified either by using absolute coordinates in the Virtual Environment or by using a named landmark. A named landmark refers to a named location previously specified in a Static Structure Diagram. This location refers to a position in the Virtual Environment. Either the absolute coordinate or the name of the landmark is given through a textual label below the landmark element. The path specifiers connect the different landmarks to each other and thereby specify the path that needs to be followed by the object involved in the pattern movement. These path specifiers can be one-way or two-way, meaning that the object can move between the two connected landmarks respectively in one direction or in both directions. There can be multiple outgoing arrows, which means that the object will randomly choose the next landmark to go to. Going from one landmark to another will be achieved through interpolation to ensure that the different pieces of the path smoothly fit together. In this way, the actor involved in the pattern will follow a smooth path.

6.4.2.5 Usage

The usage of the pattern is shown in figure 6.12. In this example, it is applied to an actor called Bus. The actor is connected to the BusBehaviour element. Within the BusBehaviour element, the path that the bus has to follow is given by means of the landmarks, named A-1, B-1, A-2 and B-2. These landmarks refer to actual positions that have been marked in the Static Structure Diagrams (not shown here). In this example, the path specifiers correspond to the streets in the virtual city application meaning that an arrow from A-1 to B-1 represents a road going from A-1 to B-1. Note that, it is the responsibility of the designer to specify meaningful connections between the different landmarks.

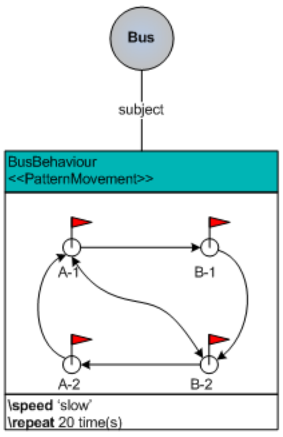


Figure 6.12: *Behaviour Definition Diagram using the Pattern Movement pattern*

6.4.3 Herd Pattern

6.4.3.1 Name

Herd

6.4.3.2 Description

The main goal of this pattern is to allow a number of objects to move as a single cohesive group (and possibly following a leader). Craig Reynolds was the first to introduce this kind of grouping behaviour in his flocking algorithms [Reynolds, 1987].

6.4.3.3 Motivation

The pattern represents a well-known type of behaviour used in many different kinds of applications. It can for example be a group of soldiers following their commandant in chief in a combat game, a group of animals in an adventure game, or a fleet of jet fighter planes in a flight simulator. In the virtual city application, the herd pattern is used to enable a number of avatars (tourists) to follow a tour guide.

6.4.3.4 Structure

The herd pattern is defined using a single behavioural element and two or more actor elements. The actors that are involved are either participating as an "item" or as a "master". This is denoted by means of a participant link towards the behavioural element of the pattern. The actors of the herd pattern are actually all the objects that belong to the group. The master represents the group leader. It is the head of the group and will thus determine the behaviour of the items, i.e. the followers. There can be multiple item-actors but only one master-actor in this pattern.

This behaviour requires the specification of the *field-of-view* parameter (the range, specified through a radius, by which the world can be observed), the *blind-spot-angle* parameter and the size of the blind spot in the field of view, which starts directly behind the item and which represents the area in which the items cannot perceive (view) anything. It actually determines the visibility for the items in the group; the larger the angle, the lesser the item can see. Furthermore, the herd behaviour needs the specification of the weights for *cohesion*, *alignment*, and *separation*. Cohesion enforces the actors to be close to the average position of the group. Alignment aims to have the orientation aligned with the rest of the group. Separation makes sure that there is a certain distance between the items of the group so that they are not colliding. Different weights for these rules can be set by the designer, which will result in the actual steering force of each of the items. Similar to the chase-evade pattern, also here, an obstacle avoidance algorithm can be specified.

6.4.3.5 Usage

There are basically two ways of using this pattern. The first way is to enumerate all the actors that are involved in the group. An example is given in the JointPatrol (figure 6.13a) behaviour where a fixed number of Guards (the item-actors) are used with one Leader (the master-actor). However, if a large number of item-actors are involved, this will quickly clutter the diagram. For this reason, a second way of using this pattern is provided which avoids listing explicitly all the item-actors. This is done by employing the list-specifier supported by our approach (given by the {. . . }* notation) as illustrated in figure 6.13b, where a GuidedTour behaviour is expressed involving a set of Tourist actors (item-actors) (expressed by means of {. . . }*) which follow a Guide (master-actor). The parameter dialog for this pattern is omitted.

6.4.4 Strategy Pattern

6.4.4.1 Name

Strategy

6.4.4.2 Description

Many applications are often faced with the issue of selecting a particular behaviour in a particular context. The strategy pattern aims at allowing the designer to specify how or when the appropriate behaviour should be selected. It allows defining a family of behaviours, all having the same

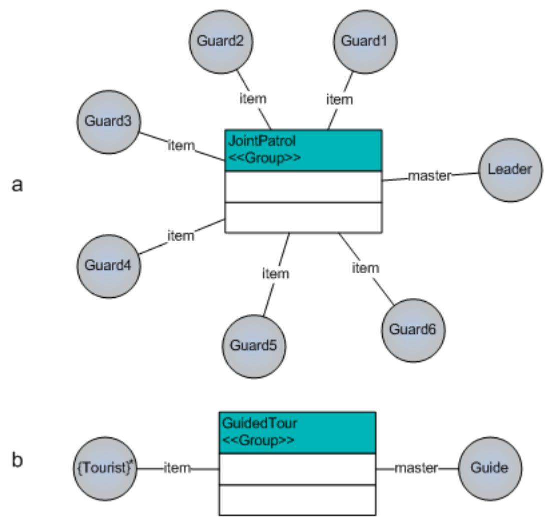


Figure 6.13: *Behaviour Definition Diagram using the Herd pattern*

purpose, and specifying how an object at runtime can decide which one to choose.

6.4.4.3 Motivation

An example can be found in games, where an attack is done differently depending on the weapon that is carried or where a different kind of movement is performed depending on the health of the player (e.g., running when healthy, limping when hurt) or depending on the location in the world (e.g., walking when on ground, swimming when in water). For example in our virtual city application, this pattern can be used for the wandering behaviour of the avatar representing a tourist. Depending on his level of tiredness and his interest in the things around him, we want to adapt the way in which he wanders.

6.4.4.4 Structure

The strategy pattern consists of a generic behaviour, which can be considered as an *abstract strategy* that needs to be replaced by a concrete behaviour when invoked. The abstract strategy is connected to a subject, being the actor that will actually be performing the behaviour, through a participant link. A number of *concrete strategies* (behaviours) are connected to the abstract strategy by means of causality links. The causality on the links is given by means of conditions. When the abstract strategy is invoked, the concrete strategy that will be executed will depend on the values of the conditional expressions that have been specified. The behaviour associated with a true condition will be executed. When more than one condition would result in a true value, one of these strategies is randomly picked. If none of the conditions are satisfied, the behaviour is set to the default one defined in the abstract strategy. The condition can be specified by means of relational operators ($<$, $>$, $<=$, $>=$, $==$). In addition, the standard arithmetic operators may be used ($+$, $-$, $/$, $*$). The conditions can also be combined or negated using the boolean operators (AND, OR, NOT). The properties of the object that is involved in the behaviour (the subject) can be referred to by means of their name. Properties of other objects or of the environment itself can be referred to using their name followed by the dot-operator (".") followed by the name of the property. The concrete strategies need to be defined elsewhere (in Behaviour Definition Diagrams).

6.4.4.5 Usage

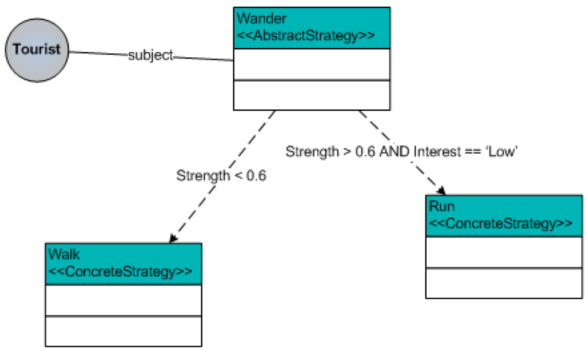


Figure 6.14: *Behaviour Definition Diagram using the Strategy pattern*

An example of how this pattern is used, is given in figure 6.14. The Tourist actor is specified as the subject of this pattern. For the abstract strategy, Wander is used. This abstract strategy is connected to the concrete behaviours Walk and Run. Depending on the strength and the general interest of the Tourist, a particular behaviour will be selected. If the Tourist is tired, he will walk slowly, if he is not tired and has a low interest, he will pass everything much faster. The concrete behaviours, Walk and Run need to be defined in other Behaviour Definition Diagrams through the standard graphical notation in our approach (see chapter 5).

6.4.5 Proxy Pattern

6.4.5.1 Name

Proxy

6.4.5.2 Description

As mentioned earlier, in our approach, events are used to specify when behaviours should be invoked for particular objects. In many cases, one wants to have a kind of indirect control to trigger the behaviour of an object as opposed to direct control in which the behaviour of an object is triggered by interacting with that object directly. With indirect control, the interaction occurs with an object different from the one of which the behaviour needs to be invoked. This pattern provides a kind of substitute behaviour that can be triggered and which on his turn triggers the desired behaviour of some other object.

6.4.5.3 Motivation

This scenario happens often in Virtual Environments. For example, some safety regulations may state that the start- and stop-button of a machine is required to be in a remote location, separate from the target machine itself. Another example can be found in an action game where the player has to pull a lever in order to open a door to the next room. In the virtual city application, this pattern was introduced on the bus stop and the bus expressing that when the user clicked on the bus stop, the bus would start its route until reaching the bus stop again.

6.4.5.4 Structure

The proxy pattern involves two different structures, the proxy-structure and the target-structure. The proxy-structure consists of an actor element, which is the subject of the pattern, connected by a participant link to a behavioural element which is the proxy. The proxy is actually the *surrogate* for the behaviour to be executed. This element is connected via a causality link to the behaviour element of the target-structure. The target-structure contains a behaviour element connected to an actor executing the behaviour. This behaviour obviously needs to be specified elsewhere in the same way as a regular behaviour is specified in our modelling language.

6.4.5.5 Usage

An instantiation of this pattern is illustrated in figure 6.15. The Lever actor participates in the pattern as the subject of the proxy element called StartMachine. This element is related to another behaviour called ProductionCycle, which is a behaviour of the actor called Machine. Here, it states that interacting with the Lever would indirectly initiate the ProductionCycle behaviour on the Machine.

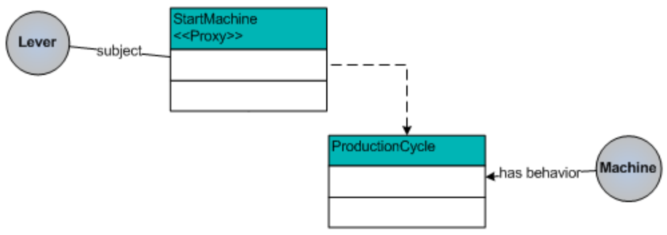


Figure 6.15: *Behaviour Definition Diagram using the Proxy pattern*

6.4.6 Randomness Pattern

6.4.6.1 Name

Randomness

6.4.6.2 Description

Much behaviour is deterministic behaviour, which may sometimes result in rather unrealistic situations. The goal of this pattern is to add a level of unpredictability to the behaviours that are executed, using the basic principles of probability and in this way giving more realism to the Virtual Environment. However, the behaviours are not completely random of course, since after a while the chances that a behaviour is executed could be known by the end-user.

6.4.6.3 Motivation

This pattern can be used to personalize a player in a game, allowing him to select an action according to his preferences or abilities. It can be used in fps games, where the computer controlling characters, i.e. the enemies, are executing some actions in a randomly fashion. In the context of our virtual city application, this pattern was applied to a number of tourists (avatars) in order to give them a kind of random behaviour by letting them to choose between different actions at runtime.

6.4.6.4 Structure

The randomness pattern is defined using a set of behavioural elements. Of this set, one is the main behaviour while the others (at least two) are the secondary behaviours. The main behaviour is causally linked to the secondary behaviours and acts as a sort of interface to them. Each of the connectors specifies the *probability* that the connected behaviour will be chosen among the set of secondary behaviours. The sum of the probabilities of all possible behaviours must be equal to 1 since at least one of the behaviours must be executed. When no probabilities are given on the connectors, it is assumed that all the behaviours have the same probability. Finally, an actor (the subject of the pattern) is connected to the main behaviour by means of a participant link. Also here, the secondary behaviours should be defined in separate diagrams using our graphical behaviour modelling language.

6.4.6.5 Usage

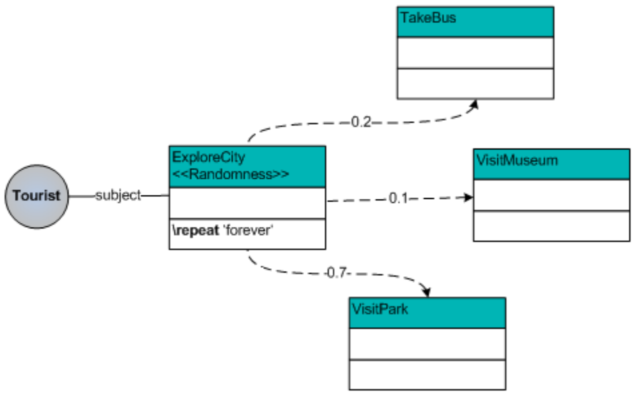


Figure 6.16: *Behaviour Definition Diagram using the Randomness pattern*

An example instantiation is given in figure 6.16. The actor, called Tourist, is the subject of the pattern and is connected to the main behavioural element, namely ExploreCity. The ExploreCity is further connected to the secondary behaviours. In this case, there is a 70 percent chance that the actor will choose the VisitPark behaviour (this high probability was taken because the park is the city's biggest attraction), 20 percent of chance that the actor will choose the TakeBus behaviour, and a 10 percent of chance that the actor will pick the VisitMuseum behaviour. In this example, the ExploreCity is set to be continuously repeated by means of the repeat command in the scripting area of the ExploreCity behaviour. This means that once the selected behaviour is finished, the actor will choose again for a next behaviour. It will execute this pattern until the behaviour is interrupted by an external event.

6.4.7 Feedback Pattern

6.4.7.1 Name

Feedback

6.4.7.2 Description

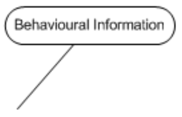
Virtual Environments are often populated with a large number of objects. End-users are often able to grasp the spatial structure and the visual properties of the objects populating the environment through exploration; however, the possible interactions and behaviours may not be obvious for the end-users. This pattern actually specifies a kind of interaction that allows the end-user to be informed about the behavioural characteristics of the objects. This may seriously improve the usability of the Virtual Environment.

6.4.7.3 Motivation

The feedback pattern can be used to inform the end-user on how to trigger the different behaviours associated with an object. In the game community, it is becoming extremely important to give visual feedback in addition to audio feedback to facilitate the game play. In the context of our virtual city application, an avatar can perform several actions, like for example, take the bus, visit the museum or visit the park. The feedback pattern can be used to inform the end-user on how to trigger these different behaviours. Another example would be feedback for the bus that is following a particular route in the city. The pattern could be used to give a textual description on the different roads that the bus is taking.

6.4.7.4 Structure

The behavioural information (feedback) to be displayed is specified by means of a *caption*, represented by means of a rounded rectangle. For this caption, a new graphical element has been introduced, because this element is conceptually different from a behavioural element and from an actor (see figure 6.17). The feedback should provide information on how an end-user can interact with the actors and information about the behaviours of the actors. The simplest way to provide the feedback, called the *behavioural information*, is by means of text (a string) in natural language. However, the behavioural information can also be specified by means of a template using replacement tags (in our notation they are enclosed by squared brackets "["..."]"). See figure 6.17 for an overview (given in BNF format) of the different tags that are available. When using the *Description* tag, an automatically generated description of the behaviour will be displayed when the behaviour is executed. See section 7.2.1 (on page 193) for more information regarding the automatic text generation. Through the *Attribute* tag, the designer can ask to visualize the value of some of the attributes of the actor or behaviour. The attribute tag is specified by means of the *Name* tag (which will return the name of the attribute) and the *Value* tag (will return the value of the attribute). Furthermore, the functionality provided by the actor can be displayed by using the *Function* tag. This tag is specified by means of an *Action* and a *Reaction* tag, resulting in the display of tuples of respectively the trigger to be used and the behaviour triggered. One can refer to a particular attribute or to a particular behaviour by explicitly mentioning the name within the replacement tags. Finally, *separators* can be used to denote how the different elements must be separated in case of multiple information items or multiple behavioural information entries. A caption element is connected to either an actor or a behavioural element by means of a participant link. Note that in case of a complex behaviour, these caption elements can also be attached to sub-behaviours within the complex behaviour. The information described in the caption element will be displayed when the end-user selects the object with the virtual pointer, or when the behaviour is executed.



Behavioural Information Description Tags

```
tags ::= { (character|digit) - '"'}
       | '[' Description ']'
       | '[' tag ']' (separator '[' tag ']')

tag ::= 'Attribute' [ '.' identifier ]
            '.' ('Name' | 'Value')
      | 'Function' [ '.' identifier ]
            '.' ('Action' | 'Reaction')

separator ::= '=' | ':' | '->'
```

Figure 6.17: *Additional graphical elements for the Feedback pattern*

6.4.7.5 Usage

The use of this pattern is shown in figure 6.18. In 6.18a the functionality of the Tourist actor will be displayed by showing the name of the triggers and the name of the behaviour executed via this trigger as follows: OnKey-Press(b) ~> TakeBus. It also illustrates the use of a separator (e.g., ~>). In 6.18b, the attribute value of one particular attribute of the Tourist object will be returned, namely that of "Strength". By the specification given in figure 6.18c, the automatically generated description of the ProductionCycle behaviour will be returned.

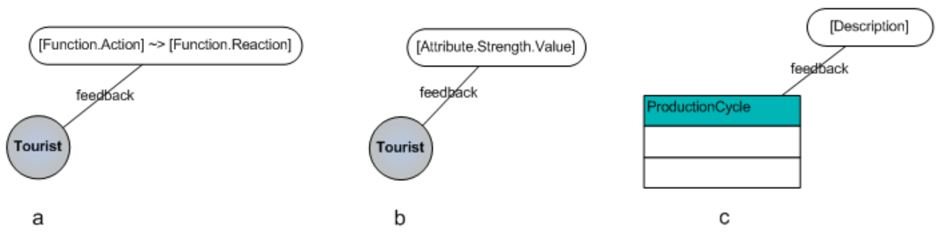


Figure 6.18: *Behaviour Definition Diagram using the Feedback pattern*

6.4.8 Device Configuration Pattern

6.4.8.1 Name

Device Configuration

6.4.8.2 Description

With the large range of input devices available in interactive applications (such as mouse, keyboard, space mouse, controllers, and so on) it may be necessary to adapt an application in such a way that the most appropriate input device can be used for invoking a behaviour. This can be realized with the device configuration pattern. This pattern is a so-called interaction pattern that allows the designer to select a particular device and reconfigure its buttons appropriately. It allows not only to easily switch between different devices in the behaviour specifications but also to simplify the incorporation of new devices.

6.4.8.3 Motivation

Many games nowadays come with controllers having a wide variety of buttons. Several combinations of buttons could be set by the player to execute a particular action. The different buttons of a space orb or space mouse [Burdea and Coiffet, 2003] might be configured for a particular Virtual Reality application. This is very important for people that are physically not able to use a particular device, in which case game actions can be mapped onto specific keys or buttons of input devices. In our virtual city application, this pattern was used to provide keyboard shortcuts for a number of behaviours to be executed.

6.4.8.4 Structure

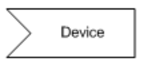


Figure 6.19: *Additional graphical element for the Device Configuration pattern*

This pattern is composed of one *device* element representing a particular physical device. This element is connected with a number of event elements. The event elements are on their turn connected to the behaviours that they need to trigger (which is the normal procedure in our behaviour modelling approach). On the connector between a device element and an event element, the designer can specify some arguments denoting the information that should be sent to the event. This can be the keys or buttons that are pressed, possibly together with the masks for any key (or button) combinations. This can also be position and orientation information. Please remember that in our approach, the definition of a behaviour is separated from the events triggering it, which is specified in a Behaviour Invocation Diagram. A device configuration pattern is therefore used within a Behaviour Invocation Diagram.

6.4.8.5 Usage

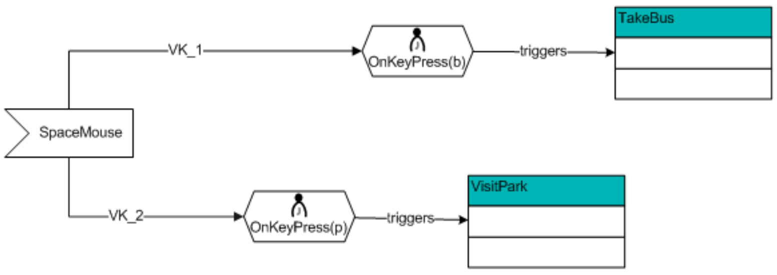


Figure 6.20: *Behaviour Invocation Diagram using the Device Configuration pattern*

Figure 6.20 gives an example use of this pattern. The device element, namely the SpaceMouse, is connected to a number of events (OnKeyPress). The different behaviours (TakeBus and VisitPark) will be executed respectively by pressing the 1-key and the 2-key on the SpaceMouse as specified by the arguments on the connectors. Key combinations can also be specified (such as ctrl+vk_b, alt+vk_enter and shift+vk_p).

6.4.9 Other Patterns

The collection of patterns presented in this section is far from complete and many other patterns could be added. For instance, a pattern that could be useful would be to provide the possibility of adding and removing behaviours at runtime where an object could gain or loose some capabilities depending on its status or on its location in the Virtual Environment. Other patterns could be, for example, patterns dealing with objects learning about their environment or with communication between objects.

Since the focus of this dissertation is on the modelling of behaviour in a Virtual Environment, only behavioural patterns were developed so far. However, it is our strong believe that the concept of visual generative design patterns can also be used in the context of the Static Structure Diagram (the diagram specifying the static structure of the Virtual Environment) in our approach. Examples could be: objects placed according to some predefined pattern (such as the setup of the players in a soccer team), or rooms connected to each other in a certain way (predefined common structures of building layouts).

6.5 Summary

The work presented in this chapter extends the work introduced in the previous chapter. It describes the second major contribution of this dissertation. Within this chapter, an extension on the behaviour modelling approach is explained. The extension combines our graphical notation with the concept of so-called visual generative design patterns. A behavioural design patterns framework, implementing the proposed approach, has been discussed. It allows a more experienced designer to create new visual generative design patterns and to include them in the framework for further use. The design patterns created with this approach are generative so that by using our framework, automatic code generation for the Virtual Environments specified is possible.

The pattern creation process can be summarized as follows. The pattern specification is used to describe the pattern from the viewpoint of our graphical notation (using graphical elements and parameters) (step (1)). In the extension specification (step (2)), the semantics of the visual pattern is specified so that the pattern can be used by the VR-WISE Conceptual Designer and the parameters can be interpreted correctly. After step (1) and (2) the specified pattern is available in the VR-WISE Conceptual Designer and can be used by the designer and applied in the design of a Virtual Environment. The use of a pattern can be done completely through our graphical notation. The executable pattern code, which is the result of the source code specification (step (3)), is compiled together with the output of the other specifications made with the visualization framework to come to a runtime binary. The result from the VR-WISE Conceptual Designer is then used as an input for this binary, which makes the actual Virtual Reality application.

The goal of this approach was twofold. On one hand, it was aimed to reduce the size and complexity of the behaviour specifications, which is considered as the main drawback of existing specification approaches such as scripting languages and graphical notations. On the other hand, it allowed us to capture existing expertise in a behaviour specification in a well-defined way and reuse it in other designs, which may result in a reduction of the time (and cost) needed for developing behaviours.

This chapter furthermore described a number of patterns created with our framework, such as patterns applying AI techniques (Chase-Evade and Herd), general-purpose behavioural patterns (Pattern Movement, Randomness, Strategy and Proxy), and some interaction patterns (Feedback and Device Configuration). As already mentioned, this collection of patterns can be extended with new patterns. It was not the intention to create a complete set of design patterns; the main intention of this collection is to show the capabilities of the behavioural design patterns framework and hereby prove its feasibility.

CHAPTER 7

Implementation

So far, this dissertation has been devoted to the general architecture of our approach, called VR-WISE, as well as how this approach was extended to enable the modelling of behaviour. The previous chapters mainly discussed the details of the behaviour modelling approach that was introduced. To show the feasibility of our approach, proof-of-concept software tools supporting the approach have been developed. The topic of this chapter is to discuss these tools and in particular the tools developed in the context of this dissertation.

The chapter will start (section 7.1) by giving a general overview of the different tools that were developed to prove the feasibility of the ideas presented in this dissertation. This section also discusses the relationships between those tools. Section 7.2 gives details on the tool developed to support the graphical modelling of the behaviour in a Virtual Environment. Section 7.3 will then go into detail on the main software application, called OntoWorld, developed to support the overall VR-WISE approach and how it was extended with the behaviour modelling approach. Afterwards, section 7.4 discusses some issues related to our viewer framework designed to visualize the end result. Finally, section 7.5 briefly summarizes this chapter.

7.1 Overview

In order to validate, and later on evaluate, the expressiveness (i.e. Is the information captured by means of the models sufficient to describe an actual Virtual Environment?) and applicability (i.e. Can a working Virtual Environment be generated from these models?) of the modelling concepts developed for the VR-WISE approach and in particular for the behaviour modelling approach, a number of prototype tools have been implemented.

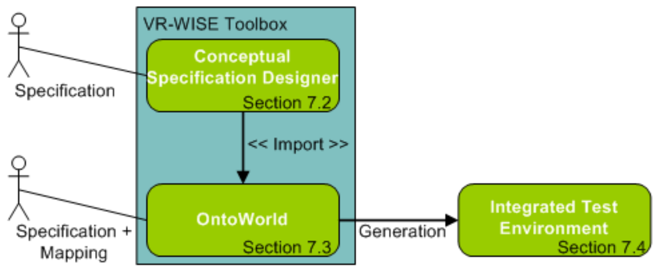


Figure 7.1: *Overview of implementations*

Figure 7.1 gives an overview of the different tools developed to support the complete VR-WISE approach, including the behaviour modelling approach. There are basically three tools:

- **Conceptual Specification Designer (CSD).** This diagram editor allows the designer to specify the high-level conceptual specifications using the graphical notation presented in previous chapters.
- **OntoWorld.** This is the main application supporting the different phases (specification, mapping and generation) of the overall VR-WISE approach through an intuitive GUI interface. The diagrams created by the first tool can be imported in this tool to replace the textual specification provided for the specification phase.
- **Integrated Test Environment (ITE).** This is a visualization framework for loading the Virtual Environment (for the static part) and processing the generated source code (for the dynamic part), coming from the OntoWorld application, in order to show the resulting Virtual Reality application.

These tools will be discussed in more details in the following sections. Only the first two are actually used for designing the Virtual Reality application and are grouped into the VR-WISE toolbox. The third one is used to visualize the end result but could be replaced by some other viewer.

7.2 Conceptual Specification Designer (CSD)

A first and important tool in the VR-WISE toolbox is the **Conceptual Specification Designer** (CSD). It is a diagram editor with a graphical interface, implemented on the .NET platform. It allows creating the high-level conceptual specifications in a graphical way (see figure 7.2). The different models describing the behaviour definition as well as the behaviour invocation can be specified using this CSD. Figure 7.3 gives a schematic overview of how this prototype is implemented.

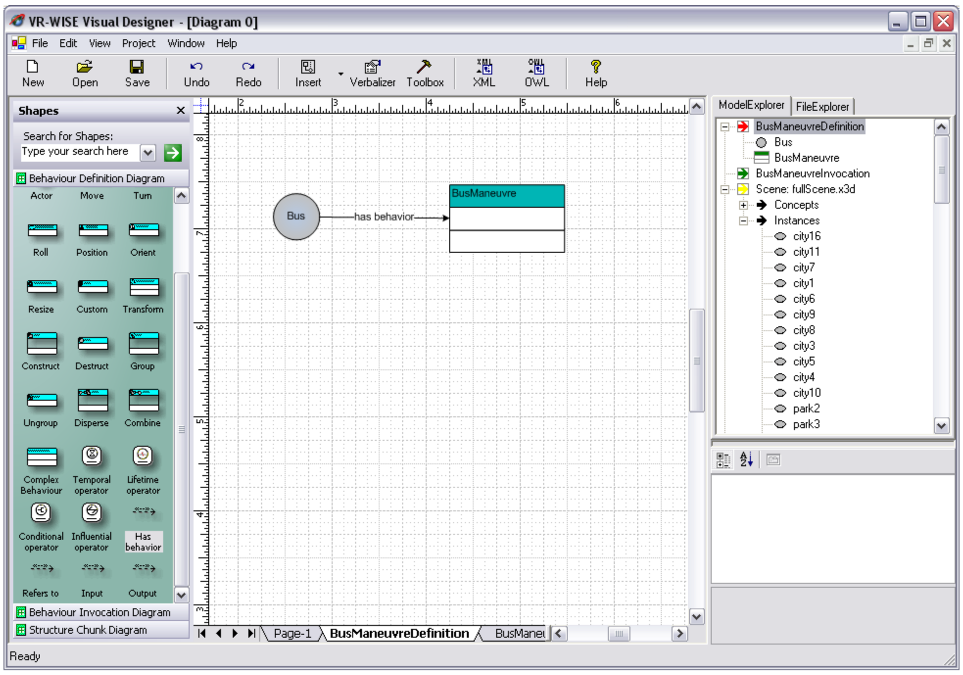


Figure 7.2: *Conceptual Specification Designer (screenshot)*

At the **Core** of the application, an internal record is kept of all the diagrams that are created (and loaded) at a particular moment in time. All this information is kept in a data structure that is called the *semantic graph*. This information can be written to a file using an XML format that can later on be imported in the OntoWorld tool. More details on the semantic graph and the OntoWorld tool are given in section 7.3. In terms of the **GUI**, it can be said that the application is MDI[1]-based. To facilitate the design of the behaviour, a so-called model explorer is available (see figure 7.2 on the right). The model explorer gives an overview of the diagrams that are currently loaded and the hierarchical structure of the graphical elements within these diagrams. Furthermore, the model explorer eases some tasks in the behaviour modelling approach. As mentioned earlier, there are places in the diagrams where 'connections' should be made between concepts from different diagrams. For example, an object (either a concept or an instance described in the Static Structure Diagrams) in a Behaviour Invocation Diagram needs to be assigned to one or more actors from the Behaviour Definition Diagram. Using the model explorer, the designer does

[1]Multi Document Interface

not need to manually specify the names for the actors assigned to the object. Instead, this can be easily done by just dragging the actor from the model explorer to the object element in the diagram. For the GUI, the application makes use of Microsoft Visio [Wideman, 2003] for the diagramming capabilities itself. In figure 7.2, this module is marked in red since it is not built by us.

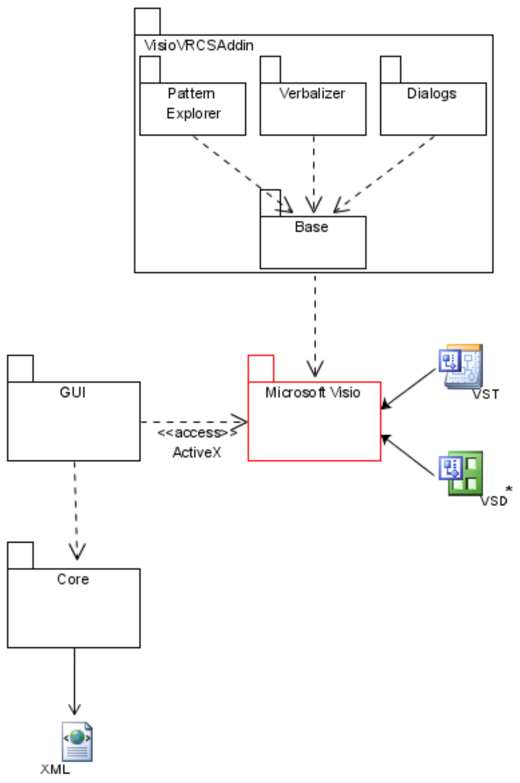


Figure 7.3: *Conceptual Specification Designer overview*

The CSD accesses Visio using an ActiveX control[2]. This enables our application to add the functionality of Visio as if it was a regular part of our application. Microsoft Visio is a drawing and diagramming program that allows quickly and easily visualizing and communicating information through diagrams.

To use Visio, a number of *stencils* have been created (see left side of figure 7.2). A stencil contains all the shapes needed for creating a particular diagram. These shapes are the representations of the graphical elements that were introduced in previous chapters; they are the key to creating diagrams in Visio. Organizing the shapes by means of the stencils makes

[2]A software module based on Microsoft's Component Object Model (COM) architecture

that they can be searched for, and referenced to, quickly. Furthermore, in this way, all the graphical elements can be easily maintained. In our approach, one stencil was created for each type of diagram (Structure Chunk, Behaviour Definition Diagram, and Behaviour Invocation Diagram) and an additional one containing all the default shapes necessary for defining the visual behavioural design patterns. By simply dragging and dropping the shapes onto the drawing page, a designer can compose a diagram.

It was also necessary to build a *template* which includes all the settings and stencils that are needed to assemble a high-level conceptual model (a combination of different diagrams) in our approach. Furthermore, the template also automatically sets up the drawing page correctly. The template together with the predefined stencils forms the basis of our application. Creating a new diagram based on the provided template initializes the second part of our CSD, namely the add-in for Microsoft Visio (see next section).

An advantage of having the CSD separated from the main application and the add-in is that the designer can work either with the normal Visio environment as well as with the CSD environment for creating the diagrams. One of the parts could also easily be replaced by some other solution without touching the other part.

7.2.1 VisioVRCSAddin (Microsoft Visio Add-in)

As soon as an instance of Microsoft Visio starts (either the application itself or through the ActiveX control from the CSD), our Microsoft Visio add-in, named **VisioVRCSAddin**, will be loaded. See the frame in figure 7.3 for a more detailed overview of the add-in.

The main part of the add-in (**Base**) includes some general functionality. There is an event handler, which is capable of catching the events coming from the Visio application, or the diagram itself. The processing of these events is also done in the add-in in order to make sure that the correct methods can be executed. Furthermore, it contains some basic classes for saving and loading the attributes required for the graphical elements of the diagram.

The bulk of the add-in consists of a collection of parameter dialogs. The graphical elements (representing the high-level modelling concepts) require a number of attributes to completely specify them. These attributes are entered through dialogs. Every element has its own dialog. When the designer double clicks the element, an event is thrown which is caught by the add-in resulting in the correct dialog being displayed. The designer can then enter the values for this particular element. Since every element has its own dialog, the dialogs remain simple.

The second part in the add-in is the **Pattern Explorer** (see figure 7.4). The purpose of the pattern explorer is on one hand to organize the different patterns available (discussed in chapter 6) and allow searching for patterns.

On the other hand it serves as a mechanism to quickly instantiate (use) a pattern. When the designer wants to instantiate a pattern from the library, all he needs to do is select the pattern in the Pattern Explorer and press the button ”insert”. After the pattern is inserted, the add-in will automatically place the graphical elements specified for the pattern on the canvas and make the appropriate connections between them, according to the *pattern description*. In other words, the designer does not need to draw the pattern manually. This tool also allows to easily manage the pattern collection, i.e. existing patterns can be modified and new patterns can be added by loading a so-called *Pattern Package*. The pattern package is an archive containing all the different files required for building a new pattern in the Behavioural Design Patterns Framework (see chapter 6). Loading a pattern package will automatically copy the files to the correct places, dynamically load the class libraries and configure the system for use of the newly added pattern.

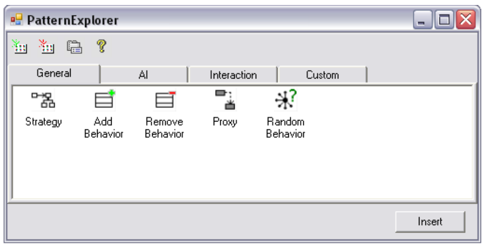
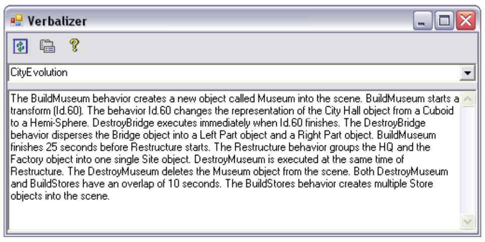


Figure 7.4: *Pattern explorer (screenshot)*

The third part of the add-in is an additional tool called the **Verbalizer** (see figure 7.5). This tool provides a natural language description of the models while the designer is making them. A template-based approach [Reiter and Dale, 1997] is used to generate the textual formulation. Every high-level modelling concept (e.g., actions, operators,. . .) is associated with a (range of) template(s). A template contains a number of slots or tags which are replaced at runtime with the correct values and this according to the parameters given for the modelling concept. A variety of templates have been created for each modelling concept, which allows us to produce text depending on the values of the parameters entered, or whether the parameters are given or not. So, any correct graphical representation can be converted into a text representation. Also for composite behaviours, text descriptions can be produced. This is based on how the composite behaviour is composed of primitive behaviours (actions).

The Verbalizer provides a mechanism to exploit the semantic information captured by means of the models. It has the following advantages:

- **Interactive Design.** Providing a natural language-like description of the behaviour that has been modelled, gives the designer a first facility to verify the model. He can check if the model indeed expresses



CityEvolution

The BuildMuseum behavior creates a new object called Museum into the scene. BuildMuseum starts a transform (Id.60). The behavior Id.60 changes the representation of the City Hall object from a Cuboid to a Hemi-Sphere. DestroyBridge executes immediately when Id.60 finishes. The DestroyBridge behavior disperses the Bridge object into a Left Part object and a Right Part object. BuildMuseum finishes 25 seconds before Restructure starts. The Restructure behavior groups the HQ and the Factory object into one single Site object. DestroyMuseum is executed at the same time of Restructure. The DestroyMuseum deletes the Museum object from the scene. Both DestroyMuseum and BuildStores have an overlap of 10 seconds. The BuildStores behavior creates multiple Store objects into the scene.

Figure 7.5: *Verbalizer (screenshot)*

what he is intending to express. This allows for an early detection of design errors. In addition, the textual descriptions may also shorten the learning time of the graphical notation.

- **Code Documentation.** After the design process has been completed, the models are used to generate programming code. The code generator can use the text descriptions, generated at design time, to document the code, and hereby facilitate the post-modelling phase and possible extensions and customization of the code.

7.3 OntoWorld

The second software application in the VR-WISE toolbox is called **OntoWorld**. The OntoWorld tool supports the overall VR-WISE approach, i.e. the creation of the conceptual models, the mappings of the conceptual level onto the implementation level, and the generation of the actual source code. Figure 7.6 gives an impression of the user interface of the tool.

To guide the user in the development process, a panel is available (on the left side of the interface) which gives an overview of the different steps comprising the VR-WISE approach. These steps correspond to the ones described in section 3.3.2. Through the user interface, the user can create a high-level specification of the Virtual Environment to be developed. This means first describing the concepts needed from the application domain under consideration (cfr. Domain Specification) and secondly defining instances of these concepts which represent the actual objects that will populate the Virtual Environment (cfr. World Specification). This can be done by means of the user interface of OntoWorld (form-based) or alternatively by making the different diagrams using the Conceptual Specification Designer and importing them into OntoWorld. Note that only the static structure can be described through the user interface of OntoWorld. Specifying the behaviours through the user interface of OntoWorld is not possible. For the behaviours, the Conceptual Specification Designer needs to be used. Then, the designer can specify the mapping for the conceptual level. This means first defining the mappings of the concepts (cfr. Domain Mapping) and then possibly override these mappings for particular instances (cfr. World Mapping). Once the specification and the mapping have been created, the code can be automatically generated by pressing the ”code generation” button.

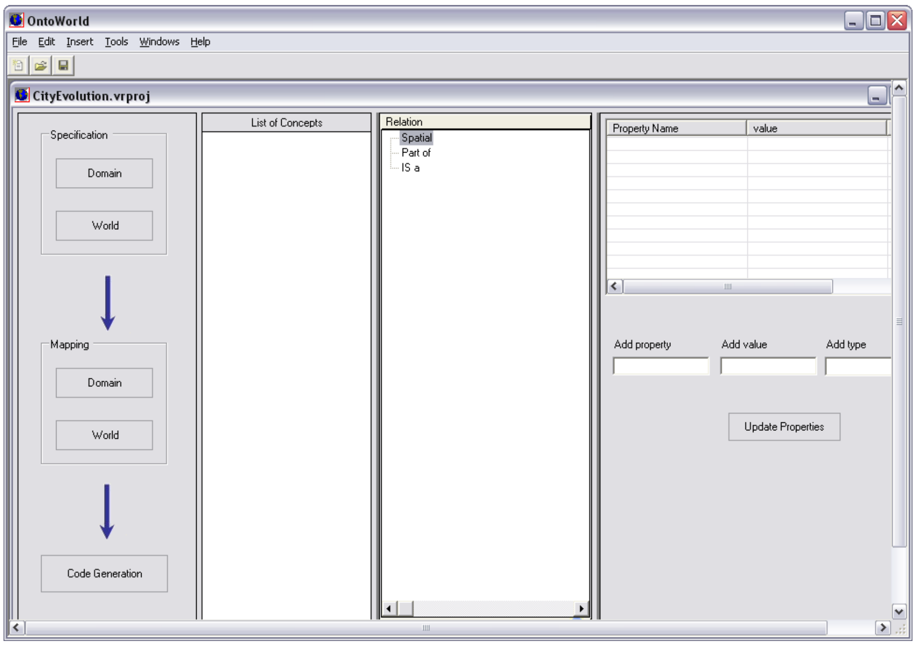


Figure 7.6: *OntoWorld (screenshot)*

Similar as the Conceptual Specification Designer, the OntoWorld tool has a MDI-based user interface. When designing this user interface, special attention was paid to the usability for the designer. If a user interface is too complex and too much functionality is offered at the same time, a non-experienced user quickly gets lost in the wide range of functionalities available. A major requirement for the design of our user interface was to provide the user at each moment a very clear picture of what needs to be done and in what stage of the development he is.

The complete user interface is contained in a single package called **GUI** as illustrated in figure 7.7. Here, an overview of the internal structure of OntoWorld is given. Next to the user interface, this tool consists of three other major packages: the Core, the Semantic Factory, and the 3D Factory.

The **Core** mainly consists of what is called the *semantic graph*. The semantic graph is the domain-specific representation of the information on which the OntoWorld tool operates. Earlier, the concept of scene graph was introduced, which is the most commonly used data structure in the Virtual Reality domain. A scene graph only contains the different nodes of the scene together with a sort of parent-child relations. The semantic graph extends the concept of scene graph in the sense that not only parent-child relations can be captured, as in the regular scene graph, but many other relations can be included as well. These relations are *is-a* relations, *part-of* relations and so on. Using this semantic graph means that practically any kind of relation could be easily added to the system. The semantic graph is a powerful data structure and much richer than a regular scene graph. The tool can save the information in the graph in an XML file and read it back later on. Furthermore, since the Conceptual Specification Designer uses the same semantic graph internally, the output of this tool can also be imported in the OntoWorld tool.

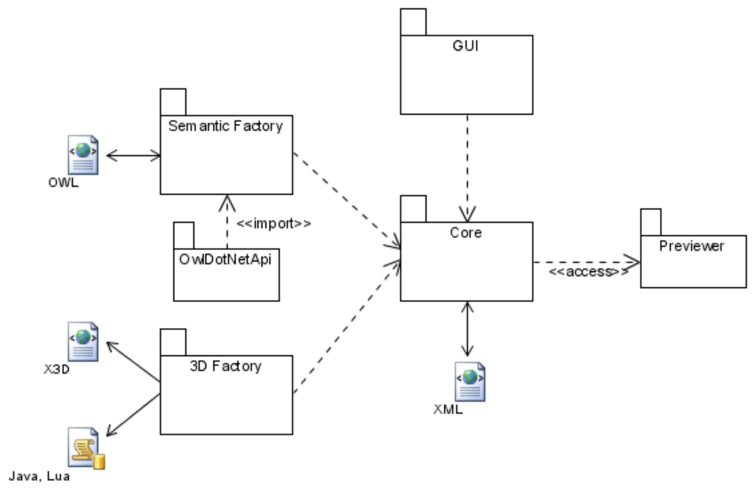


Figure 7.7: *OntoWorld overview*

The second package is the **Semantic Factory**. On the one hand, it deals with exporting the semantic information in the models to an OWL (Web Ontology Language) ontology so that this information can be used later on, for other purposes such as searching, reasoning and so on. An example of this can be found in [Mansouri, 2005]. On the other hand, the ontology format can be used to import external knowledge and use it during the design to create the high-level conceptual models. The semantic factory employs the OwlDotNetApi[3] for accomplishing this. The OwlDotNetApi is a self-developed fast and lightweight OWL API written on the .NET platform and based on the Drive RDF[4] parser. The API allows reading and writing OWL ontologies from and to a file. It furthermore allows manipulating OWL ontologies. That is, new concepts can be made; relations between those concepts can be created as well as instantiations of those concepts and so on. It is fully compliant with the W3C OWL syntax specification[5].

The third package is the **3D Factory**. This module is used to generate

[3]http://users.skynet.be/bpellens/OwlDotNetApi/
[4]http://www.driverdf.org/
[5]http://www.w3.org/TR/owl-ref/

the actual Virtual Environment. It works by transforming the conceptual specifications into a working application using the mappings. That is, for the static part, the internal semantic graph is converted into a standard scene graph. Next, all the instances (objects) are taken one by one. The behaviour specifications are searched for any of these instances playing the role of one or more actors. If this is the case, the possible behaviours belonging to the actor(s) are retrieved and programming code is generated for it (if it has not been generated before). Next, the behaviour parameterizations for the instances are added to the scene graph. The output of OntoWorld for the static part of the Virtual Environment is X3D or VRML code. The output of the dynamic part of the Virtual Environment is either Java code or Lua script. At this moment, these are the only formats that are supported. However, the package is designed in such a way that code generation towards other formats can be easily added in the future.

Please note that the OntoWorld tool is *not* a VR modelling tool. It is not capable of visually creating complex objects nor does it support advanced features such as soft body modelling, particle systems and so on. To overcome this limitation, the OntoWorld tool has been extended with an object library allowing to incorporate objects coming from dedicated VR modelling tools. The object library is a collection of (complex) objects that can be used as the target in the mapping step of the approach. These objects complement the standard primitives available in OntoWorld, and in almost any modelling application or language, such as a Box, Sphere, Cone, Cylinder. Most of the actual objects are not created by OntoWorld itself but loaded into the application from external sources. From the viewpoint of the static scene, the purpose of the OntoWorld tool is to compose the overall scene (and the objects populating it) by means of the semantic relationships that are provided in our approach.

7.4 Integrated Test Environment (ITE)

The third application that was developed is the **Integrated Test Environment** (ITE). This tool is not a part of the VR-WISE toolbox. It actually is a viewer, completely written in Java, dedicated to visualizing the output that is generated by the OntoWorld tool. Two main parts can be distinguished here.

The first part deals with visualizing the scene graph itself. For this the ITE uses Java3D[6] [Selman, 2002], a 3D graphics API for Java. The option of using *Java3D* was chosen since in this case, the application can also be loaded in an ordinary Web browser via WebStart[7]. In such a way, no applications or plug-ins need to be installed by the user in order to visualize the

[6]http://www.java3d.org/
[7]http://java.sun.com/products/javawebstart/

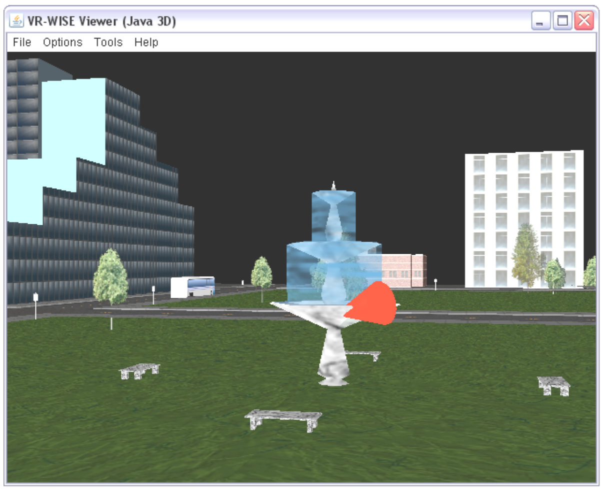


Figure 7.8: *Integrated Test Environment (screenshot)*

Virtual Environment in a Web environment. This was an important requirement since the focus of the VR-WISE approach is on Virtual Environments for the Web. It can also be executed as a local application, which is the case with the ITE. The X3D loader of Xj3D[8] is used to load the complete scene into Java3D. Furthermore, all the objects in the scene are linked to the *Ode-Java*[9] physics engine in order to give the user the feeling of being in a real physical environment. The viewer can be used in two modes, namely a navigation mode and an interaction mode. In the navigation mode, the Virtual Environment can be navigated using the standard navigational metaphors (e.g. walk, fly,...). In the interaction mode, a virtual pointer becomes visible which can be manipulated in order to interact with the objects in the Virtual Environment.

The second part deals with managing behaviours. For this, a special behaviour scheduler has been implemented which is using a constraint solver, called *Cream*. Cream[10] (Constraint Resolution Enhancement And Modules) is an open source constraint solver implemented in Java. It supports a number of optimization algorithms such as Simulated Annealing, Taboo Search and so on. An in-depth discussion about these algorithms is not the purpose of this chapter but can be found in [Russel and Norvig, 1995]. The purpose of the constraint solver is to calculate the timings in which a particular action, or behaviour, needs to become active, that is, at what time an action needs to be started and at what time an action needs to be stopped. In our approach, every action (behaviour) is expressed as an interval with a beginning time and an ending time. The operators between the actions

[8]http://www.xj3d.org/
[9]https://odejava.dev.java.net/
[10]http://bach.istc.kobe-u.ac.jp/cream/

and behaviours determine the constraints between those intervals. For example, an action A connected to action B through the temporal operator Before(5s) results in the ending time of action A being 5 seconds before the beginning time of action B. In this way, both the beginning times as well as the ending times of all the action (behaviours) can be resolved. In the current version, a Branch and Bound search algorithm is used for solving this behaviour schedule. For each scheduling block inside a behaviour, a corresponding scheduler will use the behaviour graph structure at runtime to schedule the execution of the sub-behaviours. However, errors can occur at runtime if some behaviours do not respect the global constraints. For example, if action A is before action B, action B is before C and action C needs to begin simulataneously with action A then this will result in an invalid schedule. Such errors will be detected by the behaviour scheduler that will destroy the inconsistent sub-behaviours. The overall consistency of a behaviour specification thus remains under the control and responsibility of the designer. The behaviours are attached to the scene graph only when they need to be active and are removed from the scene graph when they are finished.

The Integrated Test Environment allows a designer to quickly check if his design satisfies the requirements formulated for the application. In combination with the other tools it can be regarded as a fast prototype tool for Virtual Reality applications and to debug and improve the design.

7.5 Summary

This chapter described the implementation of the prototype tools developed to prove the feasibility of the main ideas presented in this dissertation. The chapter started by giving a general overview of the different tools and how they relate to each other. There are three tools implemented, namely the Conceptual Specification Designer, OntoWorld and the Integrated Test Environment. For all these applications, the architecture is discussed as well as the data representations used.

The OntoWorld tool was originally developed to support the complete VR-WISE approach. Through the OntoWorld tool, the user is able to specify the high-level conceptual specifications, describing the Virtual Environment at a conceptual level. Afterwards, mappings towards implementation models can be defined for the conceptual level. Finally, the actual Virtual Environment can be generated from these specifications. The internal data representation used by OntoWorld is a semantic graph. This graph does not only contain the properties about objects and relationships between the objects as usually found in a standard scene graph, but also keeps track of the additional semantic information that is typical for the VR-WISE approach, i.e. it holds information about the positional relations (spatial relations, orientation relations,...), the is-a relationships, the part-of relationships, and so on.

The OntoWorld tool allows specifying the static structure of a Virtual Environment but it does not provide a way to specify the behaviour of the objects in the Virtual Environment. Therefore, a second tool, the Conceptual Specification Designer, has been developed on top of the OntoWorld Tool. The Conceptual Specification Designer supports the behaviour modelling approach as explained in this dissertation. Based on Microsoft Visio, this tool allows the designer to graphically describe the different models involved in a behaviour specification. Furthermore, through the Conceptual Specification Designer, the designer has access to the different behavioural patterns that were defined in the behavioural design patterns framework. In this chapter, we did not discuss in depth the implementation of the behavioural design patterns framework since most of the issues concerning this topic were already presented in chapter 6.

The third tool is the Integrated Test Environment which is the framework that has been built to visualize the output of the OntoWorld tool. This tool is capable of loading the scene graph and the programming code that was generated. The visual part is managed by Java3D, while the dynamic part is managed by a scheduler based on a constraint solver. This tool is valuable for debugging and adjusting the design of the Virtual Environment.

CHAPTER 8

Validation

The main work done in this dissertation was the description of a novel approach for modelling behaviour. This approach was explained in detail together with the different high-level modelling concepts developed to specify the behaviour within this approach. Furthermore, a design patterns framework was introduced which enabled the designer to model more easily complex behaviour. The purpose of this chapter is to validate this behaviour modelling approach. Note that the approach has already been partly validated through the implementation of a number of prototype tools, such as the Conceptual Specification Designer and OntoWorld (see chapter 7). The approach has been further validated by means of a user experiment. This chapter will present the results of this experiment. In this experiment, people having no VR background had to design a number of behaviours using our proof-of-concept tool to evaluate the intuitiveness of our approach. In addition, two other case scenarios are elaborated, showing how generic the approach is.

This chapter is split into two parts. First, in section 8.1, the user experiment performed to evaluate if our approach satisfies the initial goals, is described in detail and the results are discussed. Secondly, in section 8.2, the two case studies modelled using our graphical notation, a Virtual City Park and a small case of animating a Virtual Humanoid, are presented. In section 8.3, a summary of this chapter is provided.

8.1 Experimental Results

One of the objectives for this research was to develop a modelling approach for behaviour in Virtual Reality applications that allows domain experts (usually laymen in the domain of Virtual Reality) to be more involved in the process of modelling behaviour. To achieve this, it is important that the required knowledge about programming for Virtual Reality, when using our approach, is as little as possible. This is in contrast with current practice where the behaviour is usually not modelled but programmed by hand. In other words, this objective can be restated as the aim to have people having no prior knowledge in Virtual Reality technologies start designing behaviours instead of having to rely on more experienced people. This section will present a user experiment that was performed to validate if this objective has been achieved. In particular, the goal of this experiment was to evaluate the intuitiveness of our approach and the usability of the prototype software. The results of the experiment can then be taken into account in the following versions of the modelling language and the software tools.

The experiment was executed in the context of the Virtual City Park example. This example has already been used several times in this dissertation. In chapter 6 for example, a number of behavioural design patterns have been discussed in the context of this Virtual Environment. Figure 8.1 illustrates this Virtual Environment. This Virtual Environment has been created completely using the VR-WISE Conceptual Specification Designer discussed in chapter 7. The source code was generated for it allowing us to visualize it in the Integrated Test Environment. Remember that our Virtual City Park contains a park with some monuments and is surrounded by a network of roads on which cars and buses are driving. A number of buildings are placed around the park. More details on how this Virtual Environment has been created using our toolbox can be found in [Coninx et al., 2006].

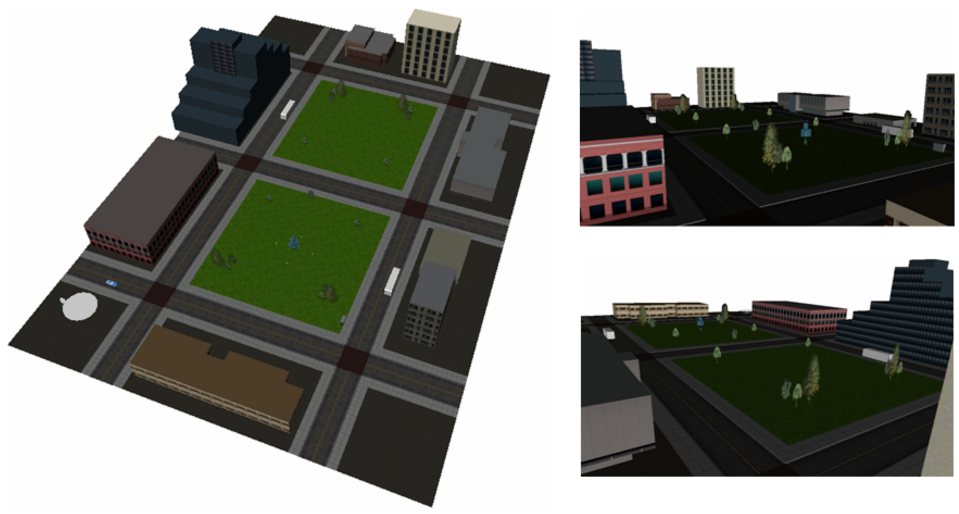


Figure 8.1: *Virtual city park*

This section is structured as follows. First, a description of the set-up of the experiment is given. Then, the data drawn from the experiment is analyzed, and the feedback received from the participants is presented. The section will finish with a discussion of the results.

8.1.1 Description

This section contains the design of the experiment, and a bit of information about the participants involved in the experiment. The materials used to help the participants performing the different tasks are also presented, and the procedure followed during the experiment is explained.

8.1.1.1 Hypothesis

In order to perform an initial evaluation of our behaviour modelling approach, we designed an experiment to test the following hypothesis: Someone with little or no experience in VR and little or no experience in conceptual modelling is able to build a valid behaviour specification using our behaviour modelling approach within a limited amount of time.

8.1.1.2 Design

The experiment consisted of the task of modelling two behaviours using our graphical behaviour modelling language. The first behaviour to model was a manoeuvre with a bus. The second behaviour to model was the evolution of a city through the time. Both behaviours were to be modelled in the context of the Virtual City Park environment mentioned earlier. To evaluate the experiment, the measuring for the different tasks included the time to complete the design of the two behaviours. In addition, the use of the correct elements for creating the behaviours was also considered.

The different behaviour modelling tasks covered the need to use a wide variety of modelling concepts. The experiment was focused on the graphical notation. The script extension was not considered, nor the ceation and usage of the behavioural design patterns.

8.1.1.3 Participants

The participants consisted of 8 people who participated individually in the experiment. They were all recruited from the Computer Science department. The experiment was done in two sessions; in each session 4 people participated. The first group consisted of one third-year bachelor student, two second-year bachelor students and one first-year bachelor student. The second group consisted of first-year bachelor students only.

All the participants were very familiar with computers and used computers at least several times a week, if not daily. The participants had, in one way or the other, been exposed to Virtual Environments (at least once) before the experiment was conducted (mostly through playing video games). None of them had any knowledge about building Virtual Environments, and especially not about modelling behaviour (dynamics) in Virtual Environments. With the exception of four participants (the second group), who only had limited programming skills, all the other participants had some knowledge about programming. Only three participants were familiar with UML or another graphical modelling language.

We believe these participants are representative for the public that we want to reach. They had no experience in any Virtual Reality technologies. They did however have some general computer science background, some programming skills but they were certainly no experienced programmers. This corresponds with the requirements that we have in mind for the people that will use the approach and the tools in practice.

8.1.1.4 Materials

This experiment was performed on four PCs of which three laptops and one desktop. The laptops were equipped with an external mouse so that the touchpad (which would make it more difficult for creating the diagrams) did not need to be used. On all the machines, the Conceptual Specification Designer tool and the Integrated Test Environment application were preinstalled.

Each of the participants received a printed introductory document, in Dutch (their mother tongue), containing a very short overview of the different steps involved in the behaviour modelling approach as well as an overview of the most important modelling concepts together with their graphical notation and a short explanation (i.e. use of the elements and parameters required for them). The document also contained a small example. This document is given in appendix C.1.

In addition, for each of the behaviour modelling tasks that the participants were asked to perform, a short video was prepared showing the desired end result of the behaviour in the Virtual Environment. These videos were available on each of the machines on which the experiment was executed and could be consulted as many times as needed during the modelling of the behaviours.

Finally, the complete static Virtual Environment (as shown in figure 8.1) was also provided as well as a ground plan (top-view) of the city containing the different objects together with their names as defined in the Virtual Environment.

8.1.1.5 Procedure

The experiment was performed in three parts. During the first part, the participants were given a twenty-five minute explanation about the behaviour modelling approach and about the different modelling concepts, following the introductory document they received. Any additional questions they had at that moment were answered. Then, a five minute course was given about the most important features of the Conceptual Specification Designer tool and the Integrated Test Environment application. This included the features for creating a new project, inserting new diagrams to this project, dragging and dropping the different elements on the canvas and generating the actual programming code. Furthermore, the navigation and the interaction with the actual Virtual Environment through the Integrated Test Environment were also explained.

The second part was the actual experiment. Each participant received a textual description (in Dutch) of the tasks to be performed (see appendix C.2).

The intention of the first exercise was to create a behaviour for a bus inside the Virtual City Park. The behaviour consisted of one manoeuvre. The bus should drive towards the next crossroad, make a left turn, stop there for a moment, and then go back and head for the bus stop a little further down the road. The Behaviour Definition Diagram of this manoeuvre is shown in figure 8.2.

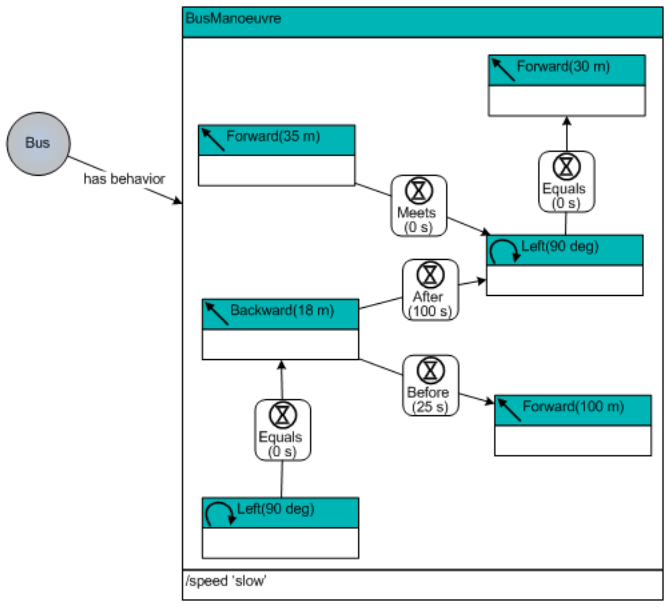


Figure 8.2: *BusManoeuvre: Behaviour Definition Diagram*

Figure 8.3 gives the Behaviour Invocation Diagram of the manoeuvre behaviour. It specifies that the behaviour is attached to the object *bus1* in our Virtual City Park and should be triggered by means of pressing a key.

The aim of the second exercise was to create an environment-behaviour,

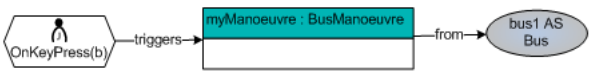


Figure 8.3: *BusManoeuvre: Behaviour Invocation Diagram*

i.e. a behaviour not attached to a single object but attached to the complete environment. The result of this behaviour would be the evolution of the city over time. Firstly, the bank building should be deleted from the scene, and while this is still happening, the building for the electricity company should transform into a more high-tech building. And finally, after some time, a new palace should be added to the scene on the left of the hotel. The Behaviour Definition Diagram of this city evolution behaviour is shown in figure 8.4.

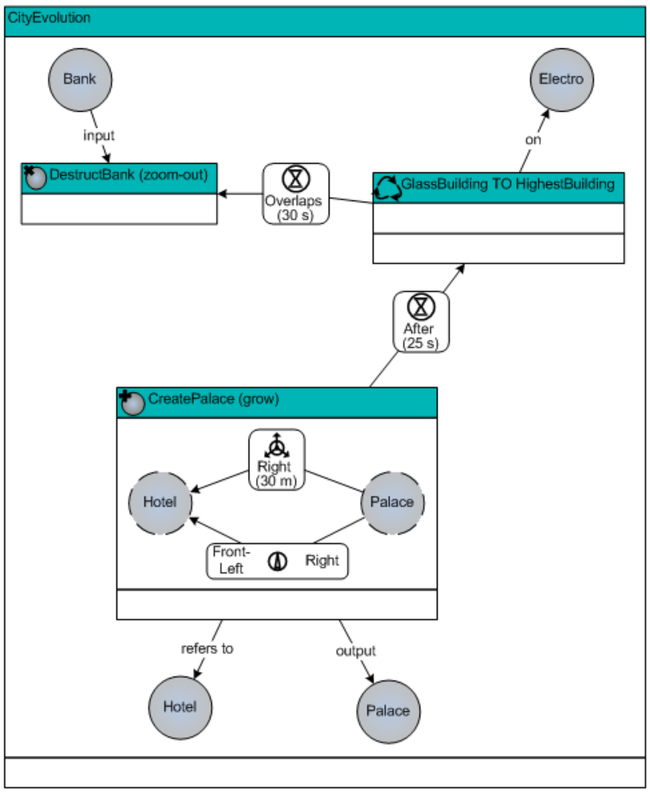


Figure 8.4: *CityEvolution: Behaviour Definition Diagram*

Figure 8.5 gives the Behaviour Invocation Diagram of the city evolution behaviour. The behaviour refers to *fortis* as the bank, *electrabel* as the building for the electricity company, *hilton* as the hotel and *bramspalace* as the palace. Furthermore, it is triggered at a particular date and time.

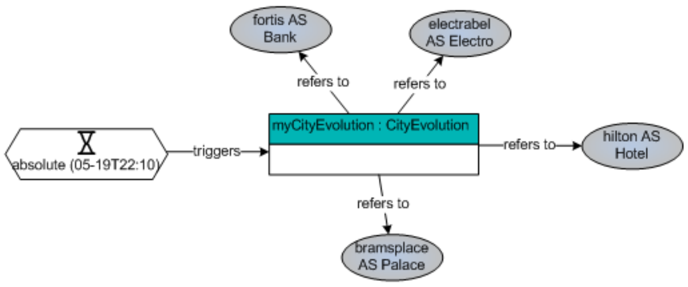


Figure 8.5: *CityEvolution: Behaviour Invocation Diagram*

The only intervention that was done during the exercises was to explain the error messages given by the software application and to answer general questions not directly related to the exercises themselves.

After performing the tasks, the participants were asked to fill in a small questionnaire (see appendix C.3). The questions were about the participant's impressions on using the graphical behaviour modelling language and the prototype software applications.

The total amount of time to complete the experiment took no more than two and a half hours.

8.1.2 Data

After each session, the diagrams made by the participants as well as the generated files were examined. Additionally, the time needed to complete the tasks was documented. Table 8.1 gives an overview of the performance of both groups separately as well as in total.

Table 8.1: *Behaviour modelling task performance*

Description		Behaviour	
		BusManoeuvre	CityEvolution
Group 1	*M*	47.25	43.75
	SD	2.98	6.24
Group 2	*M*	33.5	41.75
	SD	2.38	8.42
Total	*M*	40.375	42.75
	SD	7.76	6.94

Figures are in *minutes*; *M* (Mean); SD (Standard Deviation)

Table 8.1 shows that the participants needed an average of 40 minutes for the first exercise (BusManoeuvre) and nearly 43 minutes for the second exercise (CityEvolution). For the first exercise, the numbers are quite close to each other as can be derived from the small standard deviations (less than 3 minutes for both groups). This is not the case for the second exercise where the times needed to complete the task are more diverse as one can see from the larger standard deviations (up to 8 minutes for group 2). Furthermore, the table also shows that the second group performed their exercises considerably faster than the first group. Especially for the first exercise, this even resulted in an average difference of ±14 minutes. In the second exercise, there is also a difference but it is less distinctive (±2 minutes).

8.1.3 Feedback

Next to examining the performance, also some direct feedback was received from the participants concerning the approach, the tools, and the exercises they had to model.

In general, the participants found the approach rather intuitive. Most of the participants were quite enthusiastic, and did not have any remarks on the two-level approach that was used (the *behaviour definition* and the *behaviour invocation*).

As explained earlier, our behaviour modelling language uses icons to denote the meaning of the graphical elements. These icons were found to be intuitive by all the participants.

Concerning the exercises, two participants found the first exercise the most difficult one. The reason they gave was that, in order to start designing the behaviours, one should place oneself inside the Virtual Environment. Especially in the beginning, this led to some difficulties since the participants looked to the movements and the positioning from an external point of view, i.e. as a person looking to the Virtual Environment from the top.

Five out of eight participants found the second exercise, and in particular the *construct* action, the most difficult one. The reason for this was the Structure Chunk that had to be created to position the newly created object. Most of the participants were a bit lost in the sense that they did not know what exactly to specify there. This was firstly due to the fact that one of the item elements needed to refer to the output actor of the behaviour. Another item element needed to refer to an actor (representing an existing building) to use as a reference object in the relationships. The link between the actors of the action and the items in the Structure Chunk was confusing for the participants.

Concerning the software, two participants advised that for some structures (e.g., a particular action always requires an actor connected to it through an input link), the required elements could be automatically added to the canvas and the correct connections should be made automatically when dropping the main element. Furthermore, they missed some support for naming the different elements. A lot of elements have to be named and the overview of the different names that have already been used (or that are available to be used) is therefore lost very quickly.

8.1.4 Discussion

The performance that was measured during this experiment (see table 8.1) suggests that considering the small size of the exercises, the time needed for modelling the behaviours might be quite long. However, we should take into consideration that the participants were not familiar with modelling in general and that, as with any new technology, some time is required to get acquainted with the tools that needed to be used as well as with the behaviour modelling approach. This learning time was included in the time measured. In future experiments, an explicit learning period before the actual experiment, could be used to eliminate the influence of the learning time.

The participants were quite positive about the behaviour modelling approach. However, one has to take into account that all participants were Computer Science students. It is however possible that people having no education in Computer Science whatsoever could find this approach rather awkward since they are not comfortable with these kinds of abstraction mechanisms. As we require some background knowledge in Computer Science for using the approach, this is not relevant. However, on the other hand, we also want to use the models for communicating with other stakeholders, which may not have any Computer Science background. Therefore, it may be interesting to set up experiments to find out how easy it is, or how much time it requires for these people to understand the models.

The reason that particularly the first exercise took more time, and even was a problem for some participants is due to the fact that, as was explained earlier, the reference frame that is used in the behaviour modelling approach is different from that of a typical modelling application. All actions and relations are expressed using the local reference frame of the objects involved. This way of thinking obviously requires some adaptation of the designer. In the approach, there is the possibility to specify the actions and the positioning of the objects using an external reference frame but at the time of the experiment, this was not yet implemented. It is unclear whether this would simplify the task or instead make it even more difficult. However, there should at least be the choice to use one method or the other. Though, once the participants were more used to the approach, it became clearer to them.

The figures in table 8.1 show that the second exercise lasted longer. This was mainly due to the fact that some participants created invalid diagrams, which then took a while for them to figure out what needed to be changed to correct the models. From the feedback received from the participants, it was found that the participants considered that the second exercise was the most difficult behaviour to model. One explanation for this issue is that in the software application, a separate page is needed to specify the Structure Chunk. This makes it more difficult to keep the link between elements in the Structure Chunk and the actual graphical representation of the *construct* action itself. The relations (spatial relation and orientation relation) on itself were not a problem, although one of the participants would rather have preferred to position the object in a specific location through coordinates.

For the difference in time between the first group and the second group, no reasonable answer can be found at this moment. It is in contradiction with what was expected since the first group had more experience in Computer Science in general, and thus they could be able to solve the assignments more quickly than the second group. Maybe the motivation of this group was lower.

While examining the resulting files, it was found that for modelling the behaviours, different participants used different layouts. Some of them placed all the actions on a horizontal line while others placed them in a vertical line and others used a totally different configuration. This is not possible when writing scripting code where a concrete order (and syntax) has to be followed and from which no one can deviate. A nice issue in this regard is also the method followed by the participants to create the behaviours. Most participants immediately started decomposing the behaviour into the different actions required and placed all these actions onto the canvas first. Afterwards, the operators were selected to connect the different actions to each other.

Another interesting issue, related to the *temporal operators*, came up since the participants, at the start of the experiment, were a bit confused and often placed the arrows of the operators in the wrong direction. For example, the specification $behaviour_A - operator_{(After,x)} \rightarrow behaviour_B$ is normally read as behaviour A is executed x seconds after behaviour B. That is, the reading should be done in the direction of the arrow. But for some participants, this structure was interpreted as behaviour B is executed x seconds after behaviour A. It means that the arrows were used for ordering the actions relative to each other and the operators were used to describe the relation from the behaviour with its previous one. In other words, for them, the textual name of the operator seemed to take precedence over the direction of the arrow in the operator. At first sight, this only seemed to occur when using the inverse temporal operators since in fact the time-flow in these cases is in the reverse order than the flow of the arrows.

Although these results are based on an experiment with a small group of participants, they are nonetheless encouraging. But larger experiments are needed to confirm these results.

8.2 Case Studies

Besides the experimental results, also a number of additional cases have been elaborated mainly to show the expressiveness of our behaviour modelling language.

8.2.1 Case: Virtual City Simulation

Our first case will also be in the context of the Virtual City Park that was used during the experiment. In this case, the behaviour being modelled corresponds to the history of the city throughout time.

Figure 8.6 shows an extract of the Behaviour Definition Diagram defining the behaviours of the city and those of the buildings inside this city[1]. The *BuildMuseum* behaviour for example outputs a *Museum* and it refers to the *CityHall* actor since it needs this for positioning the Museum during the execution of this behaviour. The *BuildMuseum* behaviour is starting the *transform* behaviour on the *CityHall* actor. This behaviour will transform the *CityHall* from a cube-like (Cuboid) representation to a sphere-like (Hemisphere) representation. The transformation behaviour will trigger the *BreakBridge* behaviour as soon as it finishes. By means of the *BreakBridge* behaviour, the *Bridge* will break into two pieces. The *BuildMuseum* behaviour is executed 25 seconds before the *Restructure* behaviour, which restructures the *HeadQuarters* and the *Factory* into one big industrial *Site*. This *Restructure* behaviour is running in parallel with the *DestroyMuseum* behaviour that is removing the *Museum* from the scene. The *BuildStores* behaviour is running with an overlap of 10 seconds with the *DestroyMuseum* behaviour and is creating a list of *Store* actors. Note the script, using the Behavioural Script Language, specifying a parameterized creation of a set of actors at once.

After having defined the different behaviours and their inter-relationships, the actual objects defined for the Virtual Environment can be assigned to them through a Behaviour Invocation Diagram. Figure 8.7 shows such a diagram. The Virtual Environment consists of a number of instances. The instances are associated with actors to assign behaviours to them. In order to have the behaviours executed, they need to be invoked by means of events. Since all the behaviours are related to each other, only one needs to be triggered and all the others are executed according to the specification given in the Behaviour Definition Diagram. Here, a time event has been specified that will trigger the *BuildMuseum* behaviour 2 minutes after starting the application.

After completing the diagrams, the source code was generated and the

[1]Note that in the diagram, the different subdiagrams or scripting areas are not always shown in order to give a better overview of the overall specification. This is achieved by folding-in (or folding-out) the graphical elements using our software tool.

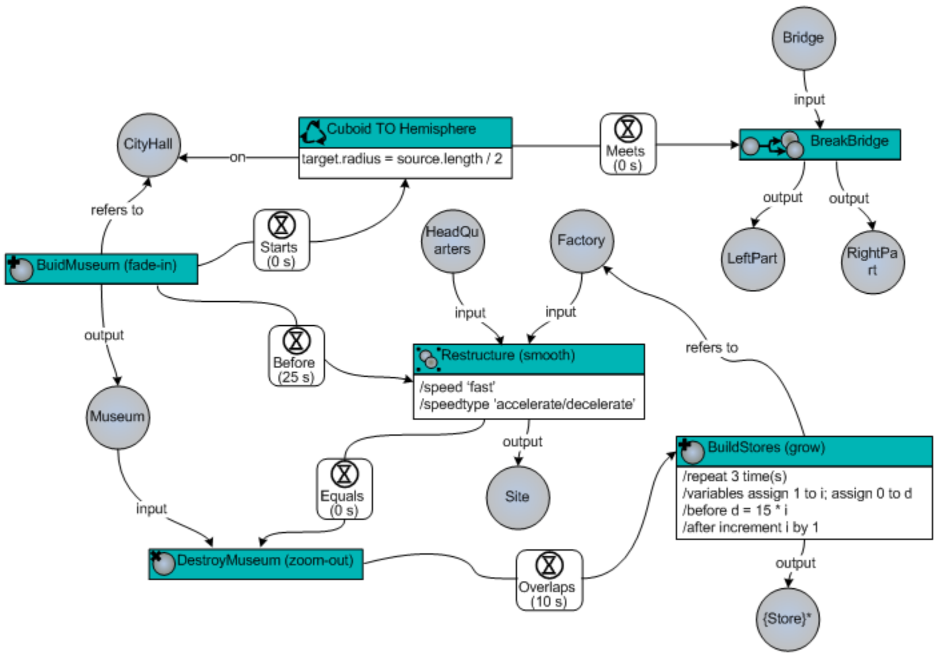


Figure 8.6: *Behaviour Definition Diagram: CityEvolution*

dynamic Virtual Environment could be inspected through the Integrated Test Environment.

8.2.2 Case: Human Animation

In this case, an example of character animation is presented which is regarded as one of toughest objects to make dynamic behaviours for. The behaviour that is modelled here is Mr. Phillip's famous back kick. Mr. Phillip is a modified version of the avatar as provided in Flux Studio[2]. Figure 8.8 shows a number of different intermediate poses that are taken by the avatar to perform this back kick. Please note that this avatar has not been created by the OntoWorld tool itself but was manually recreated and transformed into standard X3D (without H-anim[3]) to be able to load it in our Integrated Test Environment.

Figure 8.9 gives the Behaviour Definition Diagram for the back kick behaviour of our avatar. The behaviour *BackKick* is defined for the actor called *Avatar*. There are two synchronized sub-behaviours within the *BackKick* behaviour, namely *MoveBody* and *Kick*. These sub-behaviours are shown separately in figure 8.10 and 8.11. The first sub-behaviour is moving the upper part of the body, i.e. the torso, the arms, and the head (*UpperFig-*

[2]Media Machines, Inc (http://www.mediamachines.com)
[3]http://h-anim.org/

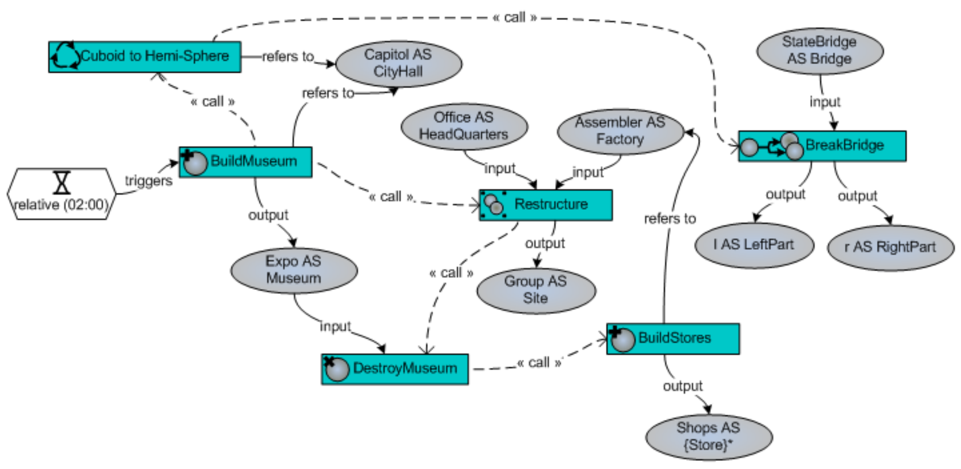


Figure 8.7: *Behaviour Invocation Diagram: CityEvolution*

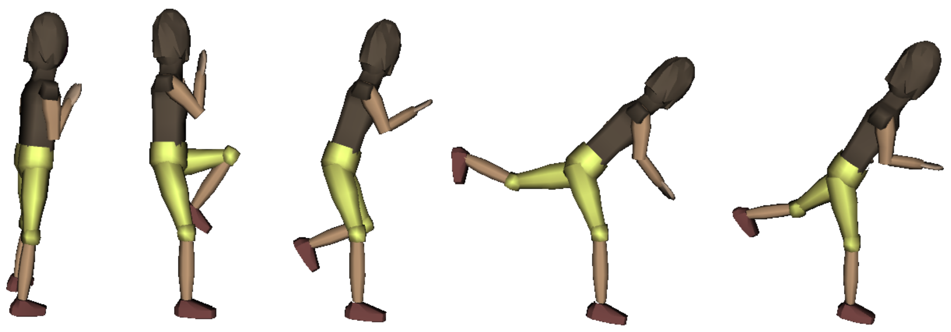


Figure 8.8: *Human animation*

ure). This behaviour is rolling the body forward which also makes that the arms (the *LeftArm* and *RightArm*) are starting to roll forward. The body is then rolling backward to its original position. The second sub-behaviour is performing the kick itself which is performed on the *Leg*. It is pulling up the *Leg* by rolling the upper leg backward while rolling the *LowerLeg* forward. Then, the leg is swung to the rear by rolling it over a very wide angle, the *LowerLeg* is also rolling simultaneously to come to a stretched position. Finally, the *Leg* is being rolled backward to an upstanding position again.

After having defined the behaviours for the avatar, the behaviours need to be associated with the actual objects (or object-parts) through a Behaviour Invocation Diagram. Figure 8.12 shows this diagram for the *BackKick* behaviour. This behaviour is linked to *MrPhillip*, the avatar in our Virtual Environment. Furthermore, the actors referred to in the behaviour are also linked to real objects as well as those that are used in the sub-behaviours. In order to have the behaviour executed, an event needs to be

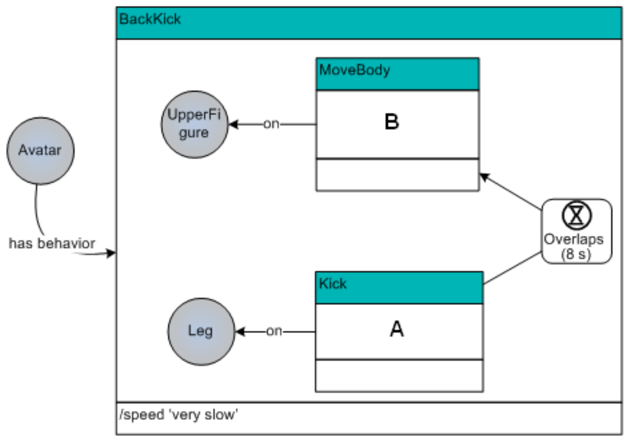


Figure 8.9: *Behaviour Definition Diagram: Human animation*

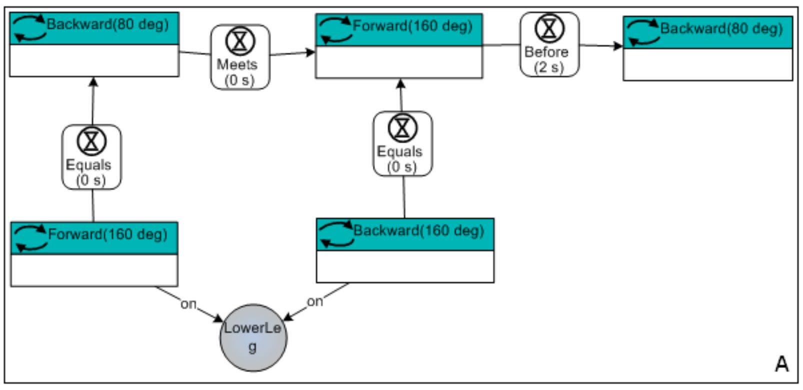


Figure 8.10: *Behaviour Definition Diagram: Human animation (con't)*

specified. Here, a user event has been specified that will trigger the *BackKick* behaviour when the avatar is touched by the user.

After having created the two diagrams, the source code could be generated and visualized in the Integrated Test Environment which gives the result as is displayed in figure 8.8.

8.2.3 Other Cases: VR-DeMo

VR-DeMo (Virtual Reality: Conceptual Descriptions and Models for the Realization of Virtual Environments) is a project that took place in collaboration with the Vrije Universiteit Brussel and Universiteit Hasselt[4]. The research done for this dissertation was part of VR-DeMo project. The project covered a wide range of research objectives dealing with most aspects of de-

[4]IWT SBO VR-DeMo (IWT 030348)

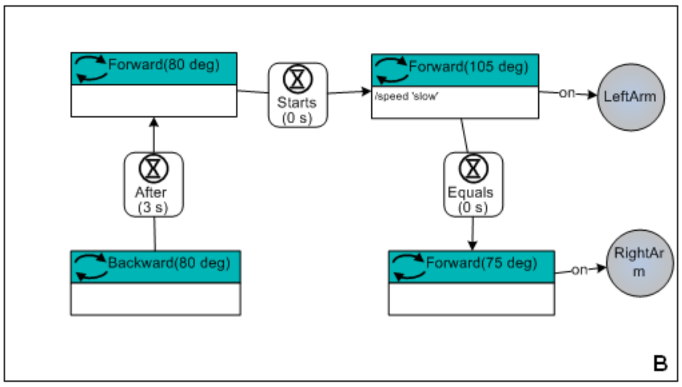


Figure 8.11: *Behaviour Definition Diagram: Human animation (con't)*

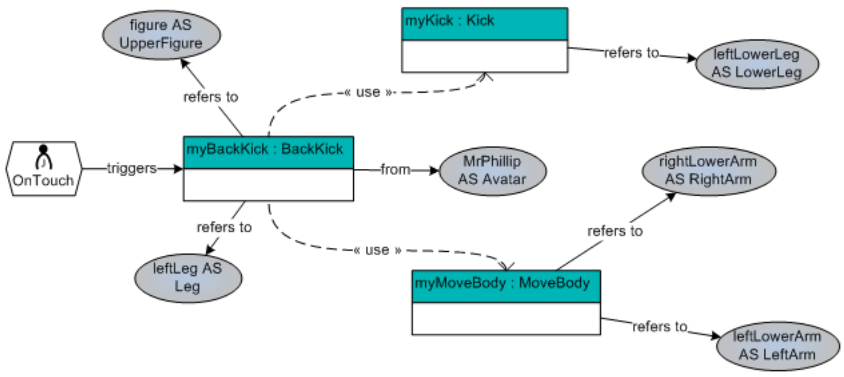


Figure 8.12: *Behaviour Invocation Diagram: Human animation*

veloping Virtual Reality applications (static scene, behaviour, interaction). In the context of this project, also a number of case studies were performed.

A large case was done for a Belgian Mining Museum, the Beringen Museum[5]. This case consisted of reconstructing the complete mining site that existed during the 20th century, above ground as well as below ground. Above ground, the site contained the two mine shafts, the docking stations, the factory, the cooling towers, the teril, and so on. Below ground, the two main tunnels were recreated. For this case, also a number of behaviours were modelled. Firstly, a behaviour was modelled to create a fly-through (by animating the camera) of the complete site and hereby construct a kind of virtual tour around the museum. Furthermore, an animation of the mining process was modelled by animating a coal wagon being driven inside the tunnels, elevated to the surface, and finally transported to the docking station. A last behaviour that was modelled for this case was the historical evolution of the mine site over time by gradually adding buildings, changing

[5]http://www.geocities.com/vlaamsmijnmuseum/ (in Dutch)

buildings, and deleting buildings.

8.3 Summary

In this chapter, the validation of our behaviour modelling approach has been described. A first validation, the implementation of the proof-of-concept tools, was discussed in the previous chapter. This chapter started by presenting in details the setup and results of a user experiment that was conducted. Then, it showed some examples of the modelled behaviours.

The first part of this chapter described a user experiment performed to evaluate the intuitiveness of our graphical notation and behaviour modelling approach. In the experiment, eight participants were given a short introduction to our behaviour modelling approach as well as on the associated modelling language. Then, they were asked to model two different behaviours in the context of an existing Virtual City Park virtual environment. The feedback that was received from this experiment will be used as input for the second version of our behaviour modelling language and tools. We do acknowledge that the experiment presented here is only a pilot study. In order to fully validate the approach, we need to do additional experiments. A first group of experiments which should be done is a comparative test between our modelling approach and other approaches. Furthermore, we can do additional experiments with people having different backgrounds, namely with and without programming background, with and without modelling background or/and with and without background in Virtual Reality to measure how the background influences the results.

To illustrate the expressiveness of our approach and of the graphical behaviour modelling language, a number of example behaviours were presented. The first example given was an extended version of the behaviour describing the evolution of the city. Last but not least, a case about human animation in which a virtual character was animated to perform a back kick was given. All these examples have been successfully modelled by means of our graphical notation and code has been generated by our software tools.

CHAPTER 9

Conclusions and Future Research

In the previous chapter, the validation of the work presented in this dissertation has been described. Firstly, a proof-of-concept implementation supporting the behaviour modelling approach was discussed. Secondly, the results of an experiment were presented. In the experiment, users had to design a number of behaviours using our proof-of-concept tool to evaluate the intuitiveness of our approach. Thirdly, a number of cases were elaborated, showing how generic the approach is. In this chapter, we look back and reflect on the work that has been done. Furthermore, we also discuss future work. We look to possible improvements but also present some new research directions that could be interesting to explore.

This final chapter is structured as follows. In section 9.1, a summary is given of the work described in this dissertation. Section 9.2, the initial example given in the introduction (section 1.2.2) is revised. Section 9.3 discusses the main contributions and achievements of this dissertation. Finally, section 9.4 presents limitations and possible future work to be done.

9.1 Summary

In this dissertation, an extension to **VR-WISE**, an existing modelling approach for Virtual Reality, is presented. The VR-WISE approach has been developed with the main purpose of facilitating and shortening the development process of Virtual Reality applications. This goal is achieved by introducing a *conceptual modelling* phase into the overall design process. The extension focuses on the modelling of behaviour in a Virtual Environment. This aspect was not yet covered in the VR-WISE approach when this PhD work started. The behaviour is one of the most difficult aspects to model and it is still not very accessible to non-VR-experts. The aim of this work was to ease the design process of behaviour through the use of intuitive high-level specifications, and without requiring advanced programming skills. In summary, this dissertation was built around three main parts, namely a *behaviour modelling approach* and the associated graphical behaviour modelling language, the *behavioural design patterns framework* and the *validation of the behaviour modelling approach.*

A behaviour modelling approach.

Specifying behaviour in this modelling approach is broken down into two consecutive steps, namely the *behaviour definition* step and the *behaviour invocation* step. The first step, the behaviour definition, allows the designer to define the different behaviours for an object. In our approach, the behaviours of an object are not only defined separated from the specification of the visual appearance of the object, but also independent of how the behaviour will be invoked. These issues are specified in the second step, the behaviour invocation. In this step, the behaviours that were defined in the behaviour definitions are assigned to the actual objects in the Virtual Environment. Furthermore, it also denotes how the behaviours may be invoked, i.e. the events that may trigger the behaviours of the objects.

Since the behaviour modelling approach is, as the overall VR-WISE approach, using *ontologies* as the underlying specification mechanism, an additional layer of abstraction has been added by means of a graphical **behaviour modelling language**. In this way, the designers do not need to be proficient in ontology languages. The behaviour modelling language involves three different kinds of diagrams: Structure Chunks, Behaviour Definition Diagrams, and Behaviour Invocation Diagrams. Each kind of diagram uses a number of high-level modelling concepts.

- A *Structure Chunk* is a small diagram used for specifying the spatial configuration of a number of objects at a particular moment in time. It describes the static scene for a small subset of objects. This is done by placing an object through spatial relations and orientation relations, defining respectively the position and orientation of an object with respect to other objects. This diagram is used within a Behaviour Definition Diagram.

- A *Behaviour Definition Diagram* is used to describe the actual behaviour of an object in the Virtual Environment. This diagram mainly contains actors (representing objects) connected to behaviours (representing the behaviours that the objects can undertake). Different types of elementary behaviours (actions) are distinguished. One set of actions focuses on the direct manipulations of the objects (e.g., move, turn, roll,...), while a second set of actions focuses on the restructuring of the scene graph (e.g., construct/destruct, group/ungroup,...). More complex behaviours can be defined by composing behaviours, either elementary or composed ones, by means of operators.

- A *Behaviour Invocation Diagram* can be considered as a kind of instantiation of a Behaviour Definition Diagram. It parameterizes and assigns behaviours defined by means of Behaviour Definition Diagrams to actual objects, being either concepts or instances populating the Virtual Environment. Furthermore, it also denotes the events that may trigger these behaviours for the particular objects.

At the beginning, our behaviour modelling language was a pure graphical language. Later, this behaviour modelling language was extended and became a kind of hybrid mix between a graphical language and a textual language. Therefore, a high-level specification language, called Behavioural Script Language (BSL), was defined. It was added to overcome the limitations of a pure graphical language and aimed at providing the designer a complementary formalism for specifying behaviours that are more complex. This language allows the designer to parameterize behaviours, to specify additional (optional) modifiers for the different actions, to specify conditions that allow to conditionally execute actions, and scripts can also be created. Scripts will be executed before, during and/or after the execution of a behaviour. This Behavioural Script Language is kept as simple and intuitive as possible.

The behavioural design patterns framework.

When modelling complex behaviour, the size of the diagrams easily becomes larger and complex. They easily reach a point where it becomes too difficult to understand and maintain them. In addition, current practice tells us that most people are trying to avoid developing complex behaviours by using and adapting existing behaviours. Therefore, the concept of *Visual Generative Design Patterns* has been applied to the modelling of behaviour in Virtual Environments. This kind of patterns distinct themselves from traditional patterns in the sense that they are described in such a way that automatic code generation becomes possible.

The framework that has been developed allows more experienced designers to create their own behavioural patterns and make them available in the graphical behaviour modelling language. The creation process follows three consecutive steps:

- In the *pattern specification*, the front-end for the graphical pattern is specified. This involves first creating the different graphical elements required by the pattern (if needed). Secondly, the actual pattern needs to be described in terms of these graphical elements, their connections, the constraints that apply as well as all the properties that are needed for the elements. Thirdly, the parameter dialogs need to be specified through an existing user interface description language.

- During the *extension specification*, the back-end for the pattern is specified. This step consists of creating a small component that will be dynamically loaded into our proof-of-concept software application. The component should contain the functionality to parse (interpret) an instance of the pattern and secondly generate the proper source code from it.
- In the *executable code specification*, the implementation of the pattern is given. This actually defines the semantics of the pattern. The implementation can be an existing implementation or it can be created from scratch. The resulting classes will be compiled together with our viewer framework to get the actual base Virtual Reality application that later on needs to be initialized.

Once a pattern has been created and added to the pattern collection, it can be used (instantiated) and applied in a particular Virtual Reality application. This process also consists of three consecutive steps. First, the appropriate pattern is selected from the collection. Then, it is adapted to suit the context of the application someone wants to develop. This can be done either via the user interfaces that have been specified for the various graphical elements or by adding and/or deleting graphical elements. Finally, the initialization code can be generated for the base Virtual Reality application.

Validation of the approach and the language.

Finally, as a validation of the principles and to show the feasibility of our approach, a prototype implementation was made; an experiment was set up; and some cases were elaborated. A graphical diagram editor has been created based on Microsoft Visio that allows the designer to visually specify the behaviour specifications using the graphical behaviour modelling language. Furthermore, the design patterns framework was implemented facilitating the creation and usage of behavioural patterns. The tools developed showed that it is possible to generate an actual dynamic Virtual Environment using the VR-WISE approach in combination with our behaviour modelling approach. This was illustrated for a number of cases, which were elaborated. Finally, an experiment that was conducted with Computer Science students using the software tool to model behaviours, has showed that the approach is indeed promising.

9.2 Modelling Behaviour and its Complexity (revised)

As a final example to illustrate the potential of the approach introduced in this dissertation, the earth system example, presented in chapter 1 (see section 1.2.2 on page 12), is reconsidered.

Although it was a simple example, it was quite laborious to specify this system in X3D. There were three main issues. Firstly, an interpolator was used which did require the use of ten different sets of values (keys and keyframes) in order to describe the smooth rotation of the objects. These values were a rough approximation and if a smoother animation was needed, more values should have been used. Secondly, two different sensors were needed to represent a single mechanism to invoke the behaviour on the objects. Thirdly, three ROUTE statements were needed to tie the sensors, the interpolator and the objects together.

Figure 9.1 presents the Behaviour Definition Diagram and the Behaviour Invocation Diagram illustrating the same behaviour specified using our graphical behaviour modelling language.

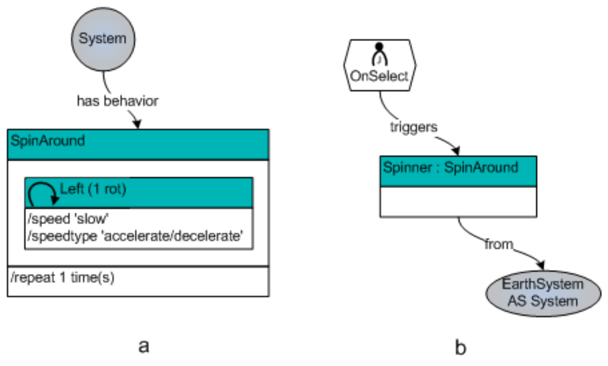


Figure 9.1: *Behaviour Definition (a) and Behaviour Invocation (b) of the EarthSystem's spinning behaviour*

Using our approach, the same behaviour can be easily defined without requiring Virtual Reality knowledge or knowledge of programming languages. As illustrated in figure 9.1a, a *SpinAround* behaviour is created for the *System* actor. This behaviour contains a single action, namely a *turn*. To have the same smooth animation as in the X3D code, the *speed* flag is set to 'slow' and the *speedtype* flag to 'accelerate/decelerate' for the turn action. Using these flags, the designer does not need to worry anymore about the different values (i.e. the keys and keyframes) for the animation. Furthermore, the *repeat* flag indicates that the behaviour should be executed only once. As shown in figure 9.1b with the Behaviour Invocation Diagram, this behaviour is connected to the target object, namely the *EarthSystem*, and a single trigger is attached to it stating that if the object is selected by the end-user (clicked), the behaviour needs to be executed. Note that the designer does not need to know any kind of Virtual Reality-specific instructions like it was the case in the specification given in chapter 1 where X3D instructions are used.

9.3 Contributions and Achievements

This dissertation contributes to the Virtual Reality domain by introducing a novel **behaviour modelling approach**. As mentioned earlier, the modelling of behaviour in a Virtual Environment is a relevant research topic since (1) the behaviour is a key element for having realistic and convincing Virtual Environments, and (2) modelling of behaviour is still difficult and not a trivial task and is often done using some dedicated scripting language directly, which makes it difficult (if not impossible) to discuss the design of the behaviours with other stakeholders in the development process. The approach presented here tried to address this issue by introducing a conceptual design phase for behaviour modelling.

Next to the fact that the approach allows the modelling of behaviour at a conceptual level, it also has a number of important additional characteristics:

- *Separation of the behaviour definition from the behaviour invocation.* This separation has two advantages: (1) it allows reuse of behaviours as the same behaviour can be attached to different types of objects as long as they satisfy the minimal requirements needed for the behaviour. (2) The same behaviour can be triggered by different interactions depending on the situation.

- *Decoupling of the dynamic aspect from the static aspect of the Virtual Environment.* Behaviours are specified independent from the specification of the static scene. This clear separation of concerns enables the designer to consider behaviour separately, thereby reducing the complexity of specifying a Virtual Reality application. Behaviour could even be (partly) specified before specifying the actual static scene. Furthermore, as the specification of behaviour is not intertwined with that of the static scene, easy reuse of behaviours for different Virtual Environments is possible.

- *Action-oriented behaviour modelling.* Action-oriented means that in the behaviour specifications, the focus is on the actions that an object needs to perform as opposed to traditional approaches focusing on the state that an object can be in. This allows for a designer not familiar with Virtual Reality implementation technology to express the behaviours in a natural way.

- *Domain independent.* The developed high-level modelling concepts for creating a behaviour specification are domain independent which make them suitable to describe any kind of behaviour for any kind of domain.

- *Generative.* The modelling concepts provided have enough expressive power so that the resulting behaviour specifications can be used to automatically generate code for them. This will result in shorter development time, as the developer does not need to write the code manually. Even in the case that very high performance is required, and the generated code would not satisfy this requirement, this facility can be used for prototyping purposes.

- *Natural language perspective.* The modelling concepts were created from a natural language perspective. This means that the way in which behaviour is described closely relates to how one would express it using natural language. This makes it easy to use and will enhance the communication with other stakeholders in the development process.

- *A mixed graphical/textual language.* By combining a graphical language with a textual scripting language, we actually combine the best of two worlds. The graphical notation allows to quickly build a mental model of a specification as the information is presented more explicitly than in case of textual scripting languages. It is easier to learn (when designed well) and may reduce the number of errors since considerably less textual input is required. The textual language on the other hand adds more expressive power to the graphical language.

Another major contribution is the **behavioural design patterns framework** which allows us to cope with more complexity while the behaviour specification remains understandable and maintainable. Using behavioural design patterns in the behaviour modelling provides us with a mechanism to explicitly capture the knowledge and expertise of a more experienced designer and allows other designers to use it in a well-defined way. Patterns also improve the reusability since a pattern can be designed and implemented once and then instantiated many times in different Virtual Reality applications. By using generative design patterns, it is also possible to use these patterns as black boxes, and to have the code generated. This will further reduce the development time, as well as the number of errors.

Another valuable contribution of this dissertation is the creation of a *behavioural pattern collection* consisting of several patterns from different categories. This demonstrates the applicability of our design patterns framework.

To realize our behaviour modelling approach, some important **research artifacts** were produced.

- An important one is the *behaviour modelling ontology.* The ontology serves as a kind of meta-model. Not only it defines the behaviour modelling language in a more formal way (i.e. what is a valid behaviour specification), but it also provides a firm basis for possible future extensions of the approach.

- A second important artifact is a set of *prototype implementations* supporting the behaviour modelling approach, being the VR-WISE Conceptual Designer and OntoWorld. Both applications are built in such a way that it is easy to change the front-end in order to cope with other graphical notations, or to change the back-end to automatically generate code for a number of other platforms.

9.4 Limitations and Future Work

In this section, we discuss the limitations of our work and indicate opportunities for improvements. Furthermore, various interesting directions for future research activities are also discussed.

The experiment performed was rather limited in different respects. First of all, only a part of the work was considered and secondly only a small amount of participants were involved. Further experiments are needed to validate if the approach indeed fulfills the expectations. For example, experiments should be performed to evaluate the Behavioural Script Language. Also, the design patterns framework needs to be carefully evaluated both on the side of pattern creation as well as on the side of pattern usage. Also, experiments with larger and more diverse groups of people are needed. Due to the small amount of participants used in the experiment, only very basic statistics could be applied. This could have led to inaccurate results. More large-scale experiments would enable us to use more advanced statistical techniques, which are better suited for handling large datasets. Considering different kinds of people could also reveal different results. We should consider people having no programming skills whatsoever as well as experienced programmers, people having no experience in the domain of Virtual Reality as well as VR-experts.

Obviously, the prototype implementations that were developed to support our behaviour modelling approach are merely proof-of-concept applications and need to be improved considerably in terms of usability. This is imperative if we want the system to be employed by a larger audience.

Also the behaviour modelling approach proposal has some limitations. Until now, the focus was put on basic manipulations of either the objects or of the scene graph. A straightforward extension of this work would be to investigate additional modelling primitives. For example, we could think of behaviour primitives for colouring and rendering as well as for sound. Also in the area of the operators, other primitives could be developed to provide complementary means of linking different behaviours. Furthermore, the existing modelling concepts are very general and aimed to be applicable to any kind of domain. It could also be interesting to look to the possibility of developing predefined modelling concepts for a particular domain. Certain domains have already established a well-defined set of conceptual modelling concepts and will prefer to use these rather than the more general ones. The modelling concepts related to these domains can be expressed in terms of the more general ones. These concepts could be seen as a specialization or refinement of the more general ones and could add more modelling power. In relation to this, also more work should be spent on extending the pattern collection. For the moment, there are only a limited amount of patterns available. This collection should be elaborated to contain a wider range of patterns.

A shortcoming in the current design pattern framework is the lack of support for pattern composition. It is known in the Design Patterns community that composition of pattern instances is something that must be considered with care when developing software applications. Also for our pattern-based framework, this is important. An actor can play several roles in different patterns and it is not always clear how all these roles interact. For example, one behavioural pattern can use another behavioural pattern or one pattern can conflict with another pattern. A conflict could occur when an actor has to perform a similar action (such as moving) in two (or more) patterns at the same time. Therefore, we should provide the designer not only a means to select and instantiate patterns (as possible now) but also a means to allow composing several patterns into one composite pattern. As mentioned above, this problem is known in the Design Patterns community and has been addressed there as well. It should be interesting to look into existing pattern composition techniques as described in [Yacoub and Ammar, 2003] and try to apply these techniques to compose behavioural patterns in the Virtual Reality domain.

An additional research activity could be to provide a tighter integration between our graphical behaviour modelling language and a graphical interaction modelling language. There is often a very thin line between behaviour and interaction, which makes it difficult to tell where the behaviour stops and where interaction takes over. Basically, our behaviour modelling approach only supports so-called *event-driven behaviour*, where the behaviour is triggered by an event and it is executed independently of the event that triggered it. However, in many cases, the interaction is much more intertwined with the behaviour in what we call *interaction-controlled behaviour* where during the complete duration of the behaviour, the user (interaction) is having control over the object. The triggering of the behaviour happens through recurring events, i.e. events are sent regularly to activate a behaviour, which checks its current state and performs the appropriate action for that time slice. A number of interaction modelling languages such as the Notation for Multimodal Interaction Techniques (NiMMiT) [De Boeck, 2007], in which interaction techniques can be described through diagrams, already exist. In order to have a better support for interaction-controlled behaviour, it is a necessity to extend our approach with some new modelling concepts. This should allow us to directly hook up to the interaction diagrams. An important aspect here is the capturing of incoming data (from external devices) and its use in the behaviours. Vice versa, it should also be possible to indicate that data should be sent back to the interaction, which could then be used to control the devices (i.e. feedback). Such an approach would definitely give an added value for both approaches (interaction modelling and behaviour modelling).

A major extension to the work described in this dissertation includes support for modelling scenarios. In the beginning of this dissertation, we claimed that a Virtual Environment consists of the static scene, behaviour, and interaction. However, this is in some cases a bit simplistic. In this way, a Virtual Environment would still be rather static, nothing would happen next to the predefined behaviours and interactions and this would not be very appealing. In practice, all Virtual Environments are designed with a goal in mind; they are part of some application, whether it is a game, a virtual shop, or something else. Therefore, there is a strong need for being able to model application-relevant scenarios. For example, in a game, the designer should be able to specify what actions the player should take to finish a level, and how his actions influence the Virtual Environment (Game World). In a virtual shop, the designer has to be able to specify which tasks the customer should go through in order to buy something [De Troyer et al., 2006]. Therefore, a kind of scenario modelling [Willemsen, 2000] [Grutzmacher et al., 2003] should be available to specify this.

Following the same motivations as for the overall VR-WISE approach, a graphical notation for modelling scenarios could be developed instead of using a scripting language. A possible solution would be to introduce two different kinds to specifications: *Actor Specification* and *Scenario Specification*.

To model a scenario, one needs to be aware of the "actors" participating in it. These actors represent the dynamic entities, either objects or users, in the Virtual Environment that can change under the impulse of the scenario. An *Actor Specification* would enable the designer to specify this. This could be done by means of a kind of elaborated Finite State Machine. It describes the way in which an actor can evolve during the execution of the scenario. In other words, the different states it can be in and the transitions between these states. Such a specification should be created for every actor that is involved in the scenario. A very important requirement here would be a tight integration with the other aspects of our modelling approach, i.e. the static scene and the behaviour. Firstly, objects can change their visual appearance throughout the scenario. This implies creating different mappings for the same object and the ability to change the mapping used during the execution of the scenario. Secondly, the objects should be able to execute behaviours (or play a role in a behavioural pattern). Mechanisms should be provided to enable the parameterization and invocation of these behaviours both inside the states as well as on the transitions.

Once all the actors involved in the scenario (and with the scenario in mind) are modelled, the actual scenario itself should be modelled, i.e. the storyline, or plot, on what could or should happen in the Virtual Environment. The main purpose of this step should be to design the scenario-specific alterations to the different Actor Specifications created previously. This could be done through a so called *Scenario Specification*. A specification like this could be specified in an advanced form of a Flow Chart, specifically tailored towards the modelling of scenarios in Virtual Environments. Two important requirements for such a specification are what we call runtime behaviour and interaction. Until now, all behaviours in our approach were indirectly linked to the objects executing the behaviours. As the execution of behaviours often depends on the course of actions taken in the scenario, this approach is not usable anymore since we do not know at design time, which objects need to execute a particular behaviour. Special constructs should be available to dynamically associate actors in the scenarios with actors in the behaviour specifications. Furthermore, interactivity needs to be taken into account as well, i.e. the Virtual Reality application should be able to cope with, or anticipate on, the unpredictability of the actions taken by the user.

We believe that an integrated development approach, namely creating a development process including static modelling, dynamic modelling and scenario modelling would present great benefits for the Virtual Reality community.

APPENDIX A

The Behaviour Modelling Ontology

```
<rdf:RDF
    xmlns:bco="http://users.skynet.be/bpellens/bco.owl#"
    xmlns:protege="http://protege.stanford.edu/plugins/owl/protege#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:time="http://www.isi.edu/~pan/damltime/time-entry.owl#"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
    xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
    xmlns:owl="http://www.w3.org/2002/07/owl#"
    xmlns="http://wise.vub.ac.be/ontoworld/bmo.owl#"
  xml:base="http://wise.vub.ac.be/ontoworld/bmo.owl">
  <owl:Ontology rdf:about="">
    <owl:imports rdf:resource="http://www.isi.edu/~pan/damltime/time-entry.owl"/>
    <owl:imports rdf:resource="http://users.skynet.be/bpellens/bco.owl"/>
  </owl:Ontology>
  <owl:Class rdf:ID="Event">
    <rdfs:subClassOf>
      <owl:Class rdf:ID="BehaviourInvocationThing"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:ID="BehaviourInvocation"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="BehaviourReference"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Object"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#BehaviourInvocation">
    <rdfs:subClassOf>
      <owl:Class rdf:about="#BehaviourInvocationThing"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#BehaviourReference"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Object"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Event"/>
  </owl:Class>
  <owl:Class rdf:ID="CompositeBehaviour">
    <owl:disjointWith>
      <owl:Class rdf:ID="Roll"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Construct"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
        <owl:onProperty>
        <owl:ObjectProperty rdf:ID="hasBehaviourThing"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:ID="Destruct"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Group"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Disperse"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Move"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Turn"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Transform"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Ungroup"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Custom"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Orientate"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:ID="Behaviour"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:ID="Combine"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Position"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Resize"/>
  </owl:disjointWith>
</owl:Class>
<owl:Class rdf:ID="Overlaps">
  <owl:disjointWith>
    <owl:Class rdf:ID="OverlappedBy"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:ID="TemporalOperator"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:ID="After"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Contains"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="During"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="EndedBy"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="StartedBy"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="hasOverlap"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:ID="MetBy"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Before"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Ends"/>
  </owl:disjointWith>
```

```
    <owl:disjointWith>
      <owl:Class rdf:ID="Starts"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Equals"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Meets"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:ID="StructureChunk">
    <owl:disjointWith>
      <owl:Class rdf:ID="Item"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:ID="Relation"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:ID="StructureChunkThing"/>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#Position">
    <owl:disjointWith>
      <owl:Class rdf:about="#Turn"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Custom"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Orientate"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Roll"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasPosition"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Behaviour"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Destruct"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Resize"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Group"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Ungroup"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Disperse"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Move"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Construct"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Transform"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Combine"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#Meets">
    <owl:disjointWith>
      <owl:Class rdf:about="#After"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#MetBy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Ends"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Equals"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#TemporalOperator"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Before"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Contains"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#EndedBy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#During"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Starts"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#StartedBy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#OverlappedBy"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Overlaps"/>
  </owl:Class>
  <owl:Class rdf:about="#During">
    <owl:disjointWith>
      <owl:Class rdf:about="#Starts"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Contains"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#After"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#TemporalOperator"/>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Meets"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasPostDuration"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasPreDuration"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#OverlappedBy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#MetBy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Before"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Equals"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#StartedBy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Ends"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#EndedBy"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Overlaps"/>
```

```
</owl:Class>
<owl:Class rdf:ID="BehaviourDefinitionThing">
  <owl:equivalentClass>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:ID="Actor"/>
        <owl:Class rdf:about="#Behaviour"/>
        <owl:Class rdf:ID="Operator"/>
        <owl:Class rdf:ID="BehaviourDefinition"/>
        <owl:Class rdf:ID="Script"/>
      </owl:unionOf>
    </owl:Class>
  </owl:equivalentClass>
</owl:Class>
<owl:Class rdf:about="#Item">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#StructureChunkThing"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#Relation"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#StructureChunk"/>
</owl:Class>
<owl:Class rdf:ID="Enable">
  <rdfs:subClassOf>
    <owl:Class rdf:ID="LifetimeOperator"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:ID="Disable"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Resume"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="Suspend"/>
  </owl:disjointWith>
</owl:Class>
<owl:Class rdf:about="#Actor">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="superActorOf"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#BehaviourDefinitionThing"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Behaviour"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#BehaviourDefinition"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Operator"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Script"/>
  </owl:disjointWith>
</owl:Class>
<owl:Class rdf:ID="OnVisible">
  <owl:disjointWith>
    <owl:Class rdf:ID="OnKeyPress"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="OnProxy"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="OnSelect"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="OnTouch"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:ID="UserEvent"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about="#Disable">
  <rdfs:subClassOf>
    <owl:Class rdf:about="#LifetimeOperator"/>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#Enable"/>
  <owl:disjointWith>
```

```
      <owl:Class rdf:about="#Resume"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Suspend"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#Resize">
    <owl:disjointWith>
      <owl:Class rdf:about="#Destruct"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Roll"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:allValuesFrom rdf:resource="http://users.skynet.be/bpellens/bco.owl#DirectionDescription"/>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasDirection"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Group"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Move"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Behaviour"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Construct"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Custom"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Position"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Turn"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasReferenceActor"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Disperse"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasMetric"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Combine"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Transform"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Ungroup"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Orientate"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:ID="ContextEvent">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasContext"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Event"/>
  <owl:disjointWith>
    <owl:Class rdf:ID="ObjectEvent"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:ID="TimeEvent"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#UserEvent"/>
  </owl:disjointWith>
</owl:Class>
<owl:Class rdf:about="#TimeEvent">
  <owl:disjointWith rdf:resource="#ContextEvent"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#ObjectEvent"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#UserEvent"/>
  </owl:disjointWith>
  <rdfs:subClassOf rdf:resource="#Event"/>
</owl:Class>
<owl:Class rdf:about="#Resume">
  <owl:disjointWith rdf:resource="#Disable"/>
  <owl:disjointWith rdf:resource="#Enable"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Suspend"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#LifetimeOperator"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="DirectionDescription">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:maxCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >3</owl:maxCardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:ID="direction"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#direction"/>
      </owl:onProperty>
      <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="http://www.w3.org/2002/07/owl#Thing"/>
</owl:Class>
<owl:Class rdf:about="#Turn">
  <owl:disjointWith>
    <owl:Class rdf:about="#Disperse"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasReferenceActor"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#Move"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Orientate"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Transform"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasDirection"/>
      </owl:onProperty>
      <owl:allValuesFrom rdf:resource="#DirectionDescription"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#Position"/>
```

```
    <owl:disjointWith>
      <owl:Class rdf:about="#Group"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Combine"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Behaviour"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Roll"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Custom"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Resize"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Ungroup"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Construct"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Destruct"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#Destruct">
    <owl:disjointWith rdf:resource="#Resize"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Roll"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Ungroup"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Combine"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:ID="hasDestructMethod"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Move"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Disperse"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Group"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasInput"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Turn"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Behaviour"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Orientate"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Transform"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Position"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Construct"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Custom"/>
    </owl:disjointWith>
  </owl:Class>
```

```
<owl:Class rdf:about="#EndedBy">
  <owl:disjointWith>
    <owl:Class rdf:about="#OverlappedBy"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Overlaps"/>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#TemporalOperator"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#StartedBy"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Equals"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Starts"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Contains"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#MetBy"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Meets"/>
  <owl:disjointWith rdf:resource="#During"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Ends"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Before"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#After"/>
  </owl:disjointWith>
</owl:Class>
<owl:Class rdf:about="#OnSelect">
  <owl:disjointWith>
    <owl:Class rdf:about="#OnKeyPress"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#OnProxy"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#OnTouch"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#OnVisible"/>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#UserEvent"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:ID="OnConstraint">
  <owl:disjointWith>
    <owl:Class rdf:ID="OnCollision"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#ObjectEvent"/>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about="#Transform">
  <owl:disjointWith rdf:resource="#Resize"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Construct"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Move"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Position"/>
  <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Orientate"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Turn"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Custom"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Ungroup"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Combine"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Behaviour"/>
  </rdfs:subClassOf>
```

```
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:ID="hasRepresentation"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Destruct"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Roll"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Disperse"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Group"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#MetBy">
    <owl:disjointWith>
      <owl:Class rdf:about="#OverlappedBy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Contains"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Starts"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Equals"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#During"/>
    <owl:disjointWith rdf:resource="#Meets"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#StartedBy"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Overlaps"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#TemporalOperator"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Ends"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#EndedBy"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Before"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#After"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#Ungroup">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasInput"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Construct"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Move"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Behaviour"/>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:ID="hasManner"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Group"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Disperse"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Custom"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Orientate"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Transform"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Combine"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Turn"/>
  <owl:disjointWith rdf:resource="#Resize"/>
  <owl:disjointWith rdf:resource="#Position"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Roll"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Destruct"/>
</owl:Class>
<owl:Class rdf:about="#Group">
  <owl:disjointWith rdf:resource="#Resize"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Combine"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Destruct"/>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Behaviour"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#Orientate"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="hasOutput"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasInput"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#Move"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Turn"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Roll"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Transform"/>
  <owl:disjointWith rdf:resource="#Ungroup"/>
  <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Disperse"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:about="#hasManner"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#Construct"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Custom"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Position"/>
</owl:Class>
<owl:Class rdf:about="#OnCollision">
  <rdfs:subClassOf>
```

```
      <owl:Class rdf:about="#ObjectEvent"/>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#OnConstraint"/>
  </owl:Class>
  <owl:Class rdf:about="#BehaviourDefinition">
    <owl:disjointWith rdf:resource="#Actor"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Behaviour"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Operator"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Script"/>
    </owl:disjointWith>
    <rdfs:subClassOf rdf:resource="#BehaviourDefinitionThing"/>
  </owl:Class>
  <owl:Class rdf:about="#StructureChunkThing">
    <owl:equivalentClass>
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#StructureChunk"/>
          <owl:Class rdf:about="#Item"/>
          <owl:Class rdf:about="#Relation"/>
        </owl:unionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>
  <owl:Class rdf:about="#UserEvent">
    <rdfs:subClassOf rdf:resource="#Event"/>
    <owl:disjointWith rdf:resource="#ContextEvent"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#ObjectEvent"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#TimeEvent"/>
  </owl:Class>
  <owl:Class rdf:about="#Equals">
    <owl:disjointWith rdf:resource="#MetBy"/>
    <owl:disjointWith rdf:resource="#During"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#TemporalOperator"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Starts"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Overlaps"/>
    <owl:disjointWith rdf:resource="#Meets"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#After"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Contains"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#StartedBy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Before"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#OverlappedBy"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#EndedBy"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Ends"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#OnTouch">
    <owl:disjointWith>
      <owl:Class rdf:about="#OnKeyPress"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#OnProxy"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#OnSelect"/>
    <owl:disjointWith rdf:resource="#OnVisible"/>
    <rdfs:subClassOf rdf:resource="#UserEvent"/>
  </owl:Class>
  <owl:Class rdf:about="#Roll">
    <owl:disjointWith rdf:resource="#Transform"/>
    <owl:disjointWith rdf:resource="#Group"/>
    <owl:disjointWith rdf:resource="#Resize"/>
    <owl:disjointWith rdf:resource="#Destruct"/>
    <rdfs:subClassOf>
```

```
      <owl:Class rdf:about="#Behaviour"/>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Ungroup"/>
    <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Disperse"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Construct"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Turn"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:allValuesFrom rdf:resource="#DirectionDescription"/>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasDirection"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Position"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasReferenceActor"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Orientate"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Move"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Combine"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Custom"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasAngle"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
  </owl:Class>
  <owl:Class rdf:about="#BehaviourReference">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="triggeredBy"/>
        </owl:onProperty>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasReference"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#BehaviourInvocationThing"/>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#BehaviourInvocation"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Object"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Event"/>
  </owl:Class>
  <owl:Class rdf:about="#Construct">
    <owl:disjointWith>
      <owl:Class rdf:about="#Custom"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Position"/>
    <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:DatatypeProperty rdf:ID="hasConstructMethod"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Group"/>
    <owl:disjointWith rdf:resource="#Turn"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Move"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasStructureChunk"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasOutput"/>
        </owl:onProperty>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Disperse"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Ungroup"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Behaviour"/>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Roll"/>
    <owl:disjointWith rdf:resource="#Resize"/>
    <owl:disjointWith rdf:resource="#Transform"/>
    <owl:disjointWith rdf:resource="#Destruct"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Combine"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Orientate"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:ID="Context"/>
  <owl:Class rdf:about="#Operator">
    <rdfs:subClassOf rdf:resource="#BehaviourDefinitionThing"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasSource"/>
        </owl:onProperty>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasTarget"/>
        </owl:onProperty>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Actor"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Behaviour"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#BehaviourDefinition"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Script"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#Move">
    <owl:disjointWith rdf:resource="#Transform"/>
    <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasDirection"/>
        </owl:onProperty>
        <owl:allValuesFrom rdf:resource="#DirectionDescription"/>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Position"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Combine"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Destruct"/>
    <owl:disjointWith rdf:resource="#Group"/>
    <owl:disjointWith rdf:resource="#Ungroup"/>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#Behaviour"/>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Disperse"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Resize"/>
    <owl:disjointWith rdf:resource="#Construct"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Custom"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Turn"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Orientate"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasReferenceActor"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Roll"/>
  </owl:Class>
  <owl:Class rdf:about="#Script">
    <owl:disjointWith rdf:resource="#Actor"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Behaviour"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#BehaviourDefinition"/>
    <owl:disjointWith rdf:resource="#Operator"/>
    <rdfs:subClassOf rdf:resource="#BehaviourDefinitionThing"/>
  </owl:Class>
  <owl:Class rdf:about="#LifetimeOperator">
    <rdfs:subClassOf rdf:resource="#Operator"/>
    <owl:disjointWith>
      <owl:Class rdf:ID="ConditionalOperator"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#TemporalOperator"/>
    </owl:disjointWith>
  </owl:Class>
  <owl:Class rdf:about="#OnProxy">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasPerimeter"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf rdf:resource="#UserEvent"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#OnKeyPress"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#OnSelect"/>
    <owl:disjointWith rdf:resource="#OnTouch"/>
    <owl:disjointWith rdf:resource="#OnVisible"/>
  </owl:Class>
  <owl:Class rdf:about="#ObjectEvent">
    <owl:disjointWith rdf:resource="#ContextEvent"/>
    <owl:disjointWith rdf:resource="#TimeEvent"/>
    <owl:disjointWith rdf:resource="#UserEvent"/>
    <rdfs:subClassOf rdf:resource="#Event"/>
  </owl:Class>
  <owl:Class rdf:about="#Behaviour">
```

```
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasScript"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf rdf:resource="#BehaviourDefinitionThing"/>
    <owl:disjointWith rdf:resource="#Actor"/>
    <owl:disjointWith rdf:resource="#BehaviourDefinition"/>
    <owl:disjointWith rdf:resource="#Operator"/>
    <owl:disjointWith rdf:resource="#Script"/>
  </owl:Class>
  <owl:Class rdf:about="#Before">
    <owl:disjointWith>
      <owl:Class rdf:about="#Starts"/>
    </owl:disjointWith>
    <rdfs:subClassOf>
      <owl:Class rdf:about="#TemporalOperator"/>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Overlaps"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#StartedBy"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#After"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Equals"/>
    <owl:disjointWith rdf:resource="#MetBy"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasDuration"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith>
      <owl:Class rdf:about="#Contains"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#OverlappedBy"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#During"/>
    <owl:disjointWith rdf:resource="#EndedBy"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Ends"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#Meets"/>
  </owl:Class>
  <owl:Class rdf:about="#Suspend">
    <rdfs:subClassOf rdf:resource="#LifetimeOperator"/>
    <owl:disjointWith rdf:resource="#Disable"/>
    <owl:disjointWith rdf:resource="#Enable"/>
    <owl:disjointWith rdf:resource="#Resume"/>
  </owl:Class>
  <owl:Class rdf:about="#Combine">
    <owl:disjointWith rdf:resource="#Turn"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Orientate"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Disperse"/>
    </owl:disjointWith>
    <owl:disjointWith>
      <owl:Class rdf:about="#Custom"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
    <owl:disjointWith rdf:resource="#Ungroup"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasOutput"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Move"/>
    <owl:disjointWith rdf:resource="#Destruct"/>
    <owl:disjointWith rdf:resource="#Construct"/>
  <owl:disjointWith rdf:resource="#Resize"/>
  <owl:disjointWith rdf:resource="#Transform"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:minCardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasInput"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Behaviour"/>
  <owl:disjointWith rdf:resource="#Group"/>
  <owl:disjointWith rdf:resource="#Roll"/>
  <owl:disjointWith rdf:resource="#Position"/>
</owl:Class>
<owl:Class rdf:about="#StartedBy">
  <owl:disjointWith rdf:resource="#EndedBy"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Ends"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Before"/>
  <owl:disjointWith rdf:resource="#Overlaps"/>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#TemporalOperator"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#After"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Equals"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Contains"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#OverlappedBy"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#MetBy"/>
  <owl:disjointWith rdf:resource="#During"/>
  <owl:disjointWith rdf:resource="#Meets"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Starts"/>
  </owl:disjointWith>
</owl:Class>
<owl:Class rdf:about="#ConditionalOperator">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:ID="hasCondition"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#Operator"/>
  <owl:disjointWith rdf:resource="#LifetimeOperator"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#TemporalOperator"/>
  </owl:disjointWith>
</owl:Class>
<owl:Class rdf:ID="OrientationRelation">
  <owl:disjointWith>
    <owl:Class rdf:ID="SpatialRelation"/>
  </owl:disjointWith>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#Relation"/>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="hasOrientation"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="http://users.skynet.be/bpellens/bco.owl#OrientationDescription"/>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasOrientation"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

```
<owl:Class rdf:about="#Ends">
  <owl:disjointWith rdf:resource="#Before"/>
  <owl:disjointWith rdf:resource="#EndedBy"/>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#TemporalOperator"/>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#StartedBy"/>
  <owl:disjointWith rdf:resource="#Overlaps"/>
  <owl:disjointWith rdf:resource="#During"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#OverlappedBy"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#After"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Starts"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#MetBy"/>
  <owl:disjointWith rdf:resource="#Equals"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Contains"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Meets"/>
</owl:Class>
<owl:Class rdf:about="#Custom">
  <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
  <owl:disjointWith rdf:resource="#Position"/>
  <owl:disjointWith rdf:resource="#Transform"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Orientate"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Construct"/>
  <owl:disjointWith rdf:resource="#Roll"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Disperse"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Destruct"/>
  <owl:disjointWith rdf:resource="#Ungroup"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:DatatypeProperty rdf:ID="hasBody"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#Move"/>
  <rdfs:subClassOf rdf:resource="#Behaviour"/>
  <owl:disjointWith rdf:resource="#Resize"/>
  <owl:disjointWith rdf:resource="#Combine"/>
  <owl:disjointWith rdf:resource="#Group"/>
  <owl:disjointWith rdf:resource="#Turn"/>
</owl:Class>
<owl:Class rdf:about="#Relation">
  <owl:disjointWith rdf:resource="#Item"/>
  <owl:disjointWith rdf:resource="#StructureChunk"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="hasReferenceItem"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#StructureChunkThing"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="hasReferredItem"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
<owl:Class rdf:about="#Starts">
  <owl:disjointWith rdf:resource="#EndedBy"/>
  <owl:disjointWith rdf:resource="#Before"/>
  <owl:disjointWith rdf:resource="#Ends"/>
  <owl:disjointWith rdf:resource="#During"/>
  <owl:disjointWith>
```

```
    <owl:Class rdf:about="#OverlappedBy"/>
  </owl:disjointWith>
  <owl:disjointWith>
    <owl:Class rdf:about="#Contains"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#MetBy"/>
  <owl:disjointWith rdf:resource="#Equals"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#After"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#StartedBy"/>
  <owl:disjointWith rdf:resource="#Meets"/>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#TemporalOperator"/>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#Overlaps"/>
</owl:Class>
<owl:Class rdf:about="#OverlappedBy">
  <owl:disjointWith rdf:resource="#Overlaps"/>
  <owl:disjointWith rdf:resource="#Meets"/>
  <owl:disjointWith rdf:resource="#Equals"/>
  <rdfs:subClassOf>
    <owl:Class rdf:about="#TemporalOperator"/>
  </rdfs:subClassOf>
  <owl:disjointWith>
    <owl:Class rdf:about="#Contains"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#StartedBy"/>
  <owl:disjointWith rdf:resource="#Before"/>
  <owl:disjointWith rdf:resource="#Starts"/>
  <owl:disjointWith rdf:resource="#Ends"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasOverlap"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#During"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#After"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#MetBy"/>
  <owl:disjointWith rdf:resource="#EndedBy"/>
</owl:Class>
<owl:Class rdf:about="#Orientate">
  <owl:disjointWith rdf:resource="#Construct"/>
  <owl:disjointWith rdf:resource="#Position"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasOrientation"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#Transform"/>
  <owl:disjointWith rdf:resource="#Combine"/>
  <owl:disjointWith rdf:resource="#Resize"/>
  <owl:disjointWith rdf:resource="#Ungroup"/>
  <owl:disjointWith rdf:resource="#Roll"/>
  <owl:disjointWith rdf:resource="#Turn"/>
  <owl:disjointWith>
    <owl:Class rdf:about="#Disperse"/>
  </owl:disjointWith>
  <owl:disjointWith rdf:resource="#Custom"/>
  <owl:disjointWith rdf:resource="#Move"/>
  <owl:disjointWith rdf:resource="#Destruct"/>
  <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
  <rdfs:subClassOf rdf:resource="#Behaviour"/>
  <owl:disjointWith rdf:resource="#Group"/>
</owl:Class>
<owl:Class rdf:about="#TemporalOperator">
  <owl:disjointWith rdf:resource="#ConditionalOperator"/>
  <owl:disjointWith rdf:resource="#LifetimeOperator"/>
  <rdfs:subClassOf rdf:resource="#Operator"/>
</owl:Class>
<owl:Class rdf:about="#BehaviourInvocationThing">
  <owl:equivalentClass>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
```

```
          <owl:Class rdf:about="#BehaviourInvocation"/>
          <owl:Class rdf:about="#BehaviourReference"/>
          <owl:Class rdf:about="#Object"/>
          <owl:Class rdf:about="#Event"/>
        </owl:unionOf>
      </owl:Class>
    </owl:equivalentClass>
  </owl:Class>
  <owl:Class rdf:about="#OnKeyPress">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:ID="hasKeyCombination"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <rdfs:subClassOf rdf:resource="#UserEvent"/>
    <owl:disjointWith rdf:resource="#OnProxy"/>
    <owl:disjointWith rdf:resource="#OnSelect"/>
    <owl:disjointWith rdf:resource="#OnTouch"/>
    <owl:disjointWith rdf:resource="#OnVisible"/>
  </owl:Class>
  <owl:Class rdf:about="#After">
    <owl:disjointWith rdf:resource="#Starts"/>
    <owl:disjointWith rdf:resource="#Ends"/>
    <owl:disjointWith rdf:resource="#Equals"/>
    <owl:disjointWith rdf:resource="#MetBy"/>
    <owl:disjointWith rdf:resource="#EndedBy"/>
    <owl:disjointWith rdf:resource="#StartedBy"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasDuration"/>
        </owl:onProperty>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Before"/>
    <owl:disjointWith rdf:resource="#Meets"/>
    <owl:disjointWith rdf:resource="#OverlappedBy"/>
    <owl:disjointWith rdf:resource="#Overlaps"/>
    <owl:disjointWith>
      <owl:Class rdf:about="#Contains"/>
    </owl:disjointWith>
    <owl:disjointWith rdf:resource="#During"/>
    <rdfs:subClassOf rdf:resource="#TemporalOperator"/>
  </owl:Class>
  <owl:Class rdf:about="#Object">
    <rdfs:subClassOf rdf:resource="#BehaviourInvocationThing"/>
    <owl:disjointWith rdf:resource="#BehaviourInvocation"/>
    <owl:disjointWith rdf:resource="#BehaviourReference"/>
    <owl:disjointWith rdf:resource="#Event"/>
  </owl:Class>
  <owl:Class rdf:about="#Disperse">
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:cardinality>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasInput"/>
        </owl:onProperty>
      </owl:Restriction>
    </rdfs:subClassOf>
    <owl:disjointWith rdf:resource="#Construct"/>
    <owl:disjointWith rdf:resource="#Transform"/>
    <owl:disjointWith rdf:resource="#Resize"/>
    <owl:disjointWith rdf:resource="#CompositeBehaviour"/>
    <rdfs:subClassOf rdf:resource="#Behaviour"/>
    <owl:disjointWith rdf:resource="#Destruct"/>
    <owl:disjointWith rdf:resource="#Position"/>
    <owl:disjointWith rdf:resource="#Move"/>
    <owl:disjointWith rdf:resource="#Roll"/>
    <owl:disjointWith rdf:resource="#Orientate"/>
    <owl:disjointWith rdf:resource="#Combine"/>
    <rdfs:subClassOf>
      <owl:Restriction>
        <owl:onProperty>
          <owl:ObjectProperty rdf:about="#hasOutput"/>
        </owl:onProperty>
        <owl:minCardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
        >1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#Ungroup"/>
  <owl:disjointWith rdf:resource="#Turn"/>
  <owl:disjointWith rdf:resource="#Custom"/>
  <owl:disjointWith rdf:resource="#Group"/>
</owl:Class>
<owl:Class rdf:about="#Contains">
  <owl:disjointWith rdf:resource="#OverlappedBy"/>
  <owl:disjointWith rdf:resource="#Starts"/>
  <owl:disjointWith rdf:resource="#EndedBy"/>
  <owl:disjointWith rdf:resource="#After"/>
  <owl:disjointWith rdf:resource="#Ends"/>
  <owl:disjointWith rdf:resource="#MetBy"/>
  <owl:disjointWith rdf:resource="#Meets"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasPostDuration"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasPreDuration"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf rdf:resource="#TemporalOperator"/>
  <owl:disjointWith rdf:resource="#Equals"/>
  <owl:disjointWith rdf:resource="#StartedBy"/>
  <owl:disjointWith rdf:resource="#During"/>
  <owl:disjointWith rdf:resource="#Before"/>
  <owl:disjointWith rdf:resource="#Overlaps"/>
</owl:Class>
<owl:Class rdf:about="#SpatialRelation">
  <rdfs:subClassOf rdf:resource="#Relation"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:ID="hasDistance"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasDirection"/>
      </owl:onProperty>
      <owl:cardinality rdf:datatype="http://www.w3.org/2001/XMLSchema#int"
      >1</owl:cardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:allValuesFrom rdf:resource="http://users.skynet.be/bpellens/bco.owl#DirectionDescription"/>
      <owl:onProperty>
        <owl:ObjectProperty rdf:about="#hasDirection"/>
      </owl:onProperty>
    </owl:Restriction>
  </rdfs:subClassOf>
  <owl:disjointWith rdf:resource="#OrientationRelation"/>
</owl:Class>
<owl:ObjectProperty rdf:ID="containsDefinitionThing">
  <rdfs:range rdf:resource="#Operator"/>
  <rdfs:range rdf:resource="#Behaviour"/>
  <rdfs:range rdf:resource="#Actor"/>
  <rdfs:domain rdf:resource="#BehaviourDefinition"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="executedOn">
  <rdfs:domain rdf:resource="#BehaviourReference"/>
  <rdfs:range rdf:resource="#Object"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasAngle">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
```

```
        <owl:Class rdf:about="#Roll"/>
        <owl:Class rdf:about="#Turn"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:range rdf:resource="http://users.skynet.be/bpellens/bco.owl#Angle"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasDuration">
  <rdfs:range rdf:resource="http://www.isi.edu/~pan/damltime/time-entry.owl#DurationDescription"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#After"/>
        <owl:Class rdf:about="#Before"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasRelativeTime">
  <rdfs:domain rdf:resource="#TimeEvent"/>
  <rdfs:range rdf:resource="http://www.isi.edu/~pan/damltime/time-entry.owl#DurationDescription"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasChild">
  <rdfs:range rdf:resource="#Actor"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasAttribute">
  <rdfs:range rdf:resource="http://users.skynet.be/bpellens/bco.owl#Attribute"/>
  <rdfs:domain rdf:resource="#Actor"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasOrientation">
  <rdfs:range rdf:resource="http://users.skynet.be/bpellens/bco.owl#Orientation"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Orientate"/>
        <owl:Class rdf:about="#OrientationRelation"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasReferenceItem">
  <rdfs:domain rdf:resource="#Relation"/>
  <rdfs:range rdf:resource="#Item"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasBehaviourThing">
  <rdfs:domain rdf:resource="#CompositeBehaviour"/>
  <rdfs:range rdf:resource="#BehaviourDefinitionThing"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="dependsOn">
  <rdfs:domain rdf:resource="#BehaviourReference"/>
  <rdfs:range rdf:resource="#BehaviourReference"/>
  <owl:inverseOf>
    <owl:ObjectProperty rdf:ID="dependentOn"/>
  </owl:inverseOf>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasParent">
  <rdfs:range rdf:resource="#Actor"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasPosition">
  <rdfs:domain rdf:resource="#Position"/>
  <rdfs:range rdf:resource="http://users.skynet.be/bpellens/bco.owl#Position"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasDirection">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Move"/>
        <owl:Class rdf:about="#Resize"/>
        <owl:Class rdf:about="#Roll"/>
        <owl:Class rdf:about="#Turn"/>
        <owl:Class rdf:about="#SpatialRelation"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasStructureChunk">
  <rdfs:range rdf:resource="#StructureChunk"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Construct"/>
        <owl:Class rdf:about="#Combine"/>
        <owl:Class rdf:about="#Disperse"/>
        <owl:Class rdf:about="#Group"/>
        <owl:Class rdf:about="#Ungroup"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="linkedTo">
  <rdfs:domain rdf:resource="#Object"/>
  <rdfs:range rdf:resource="http://users.skynet.be/bpellens/bco.owl#Concept"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasMetric">
  <rdfs:range rdf:resource="http://users.skynet.be/bpellens/bco.owl#MetricDescription"/>
  <rdfs:domain rdf:resource="#Resize"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="playsRoleOf">
  <rdfs:range rdf:resource="#Actor"/>
  <rdfs:domain rdf:resource="#Object"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#superActorOf">
  <owl:inverseOf>
    <owl:TransitiveProperty rdf:ID="subActorOf"/>
  </owl:inverseOf>
  <rdfs:domain rdf:resource="#Actor"/>
  <rdfs:range rdf:resource="#Actor"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasScript">
  <rdfs:domain rdf:resource="#Behaviour"/>
  <rdfs:range rdf:resource="#Script"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasPostDuration">
  <rdfs:range rdf:resource="http://www.isi.edu/~pan/damltime/time-entry.owl#DurationDescription"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#During"/>
        <owl:Class rdf:about="#Contains"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#dependentOn">
  <owl:inverseOf rdf:resource="#dependsOn"/>
  <rdfs:domain rdf:resource="#BehaviourReference"/>
  <rdfs:range rdf:resource="#BehaviourReference"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasPerimeter">
  <rdfs:domain rdf:resource="#OnProxy"/>
  <rdfs:range rdf:resource="http://users.skynet.be/bpellens/bco.owl#DistanceDescription"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasSource">
  <rdfs:range rdf:resource="#Behaviour"/>
  <rdfs:domain rdf:resource="#Operator"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasReferenceActor">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Move"/>
        <owl:Class rdf:about="#Roll"/>
        <owl:Class rdf:about="#Turn"/>
        <owl:Class rdf:about="#Resize"/>
        <owl:Class rdf:about="#Transform"/>
        <owl:Class rdf:about="#Construct"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:range rdf:resource="#Actor"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="containsInvocationThing">
  <rdfs:range rdf:resource="#Event"/>
  <rdfs:range rdf:resource="#BehaviourReference"/>
  <rdfs:range rdf:resource="#Object"/>
  <rdfs:domain rdf:resource="#BehaviourInvocation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasReferredItem">
  <rdfs:range rdf:resource="#Item"/>
  <rdfs:domain rdf:resource="#Relation"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasContext">
  <rdfs:range rdf:resource="#Context"/>
  <rdfs:domain rdf:resource="#ContextEvent"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasReference">
  <rdfs:domain rdf:resource="#BehaviourReference"/>
  <rdfs:range rdf:resource="#CompositeBehaviour"/>
```

```
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasDistance">
  <rdfs:range rdf:resource="http://users.skynet.be/bpellens/bco.owl#DistanceDescription"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Move"/>
        <owl:Class rdf:about="#SpatialRelation"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="containsChunkThing">
  <rdfs:domain rdf:resource="#StructureChunk"/>
  <rdfs:range rdf:resource="#Relation"/>
  <rdfs:range rdf:resource="#Item"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasOutput">
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Construct"/>
        <owl:Class rdf:about="#Group"/>
        <owl:Class rdf:about="#Combine"/>
        <owl:Class rdf:about="#Disperse"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
  <rdfs:range rdf:resource="#Actor"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasOverlap">
  <rdfs:range rdf:resource="http://www.isi.edu/~pan/damltime/time-entry.owl#DurationDescription"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#OverlappedBy"/>
        <owl:Class rdf:about="#Overlaps"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasExternal">
  <rdfs:range rdf:resource="#CompositeBehaviour"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasBehaviour">
  <rdfs:domain rdf:resource="#Actor"/>
  <rdfs:range rdf:resource="#Behaviour"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasAbsoluteTime">
  <rdfs:domain rdf:resource="#TimeEvent"/>
  <rdfs:range rdf:resource="http://www.isi.edu/~pan/damltime/time-entry.owl#CalendarClockDescription"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasInput">
  <rdfs:range rdf:resource="#Actor"/>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Combine"/>
        <owl:Class rdf:about="#Destruct"/>
        <owl:Class rdf:about="#Group"/>
        <owl:Class rdf:about="#Ungroup"/>
        <owl:Class rdf:about="#Disperse"/>
      </owl:unionOf>
    </owl:Class>
  </rdfs:domain>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasKeyCombination">
  <rdfs:domain rdf:resource="#OnKeyPress"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#triggeredBy">
  <rdfs:range rdf:resource="#Event"/>
  <rdfs:domain rdf:resource="#BehaviourReference"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="reliesOn">
  <rdfs:domain rdf:resource="#BehaviourReference"/>
  <rdfs:range rdf:resource="#Object"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasTarget">
  <rdfs:range rdf:resource="#Behaviour"/>
  <rdfs:domain rdf:resource="#Operator"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:about="#hasPreDuration">
  <rdfs:range rdf:resource="http://www.isi.edu/~pan/damltime/time-entry.owl#DurationDescription"/>
  <rdfs:domain>
```

```
      <owl:Class>
        <owl:unionOf rdf:parseType="Collection">
          <owl:Class rdf:about="#During"/>
          <owl:Class rdf:about="#Contains"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
  </owl:ObjectProperty>
  <owl:DatatypeProperty rdf:about="#hasBody">
    <rdfs:domain rdf:resource="#Custom"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasAfterSection">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Script"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasIdentifier">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#CompositeBehaviour"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasRule">
    <rdfs:domain rdf:resource="#Transform"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:about="#hasRepresentation">
    <rdfs:domain rdf:resource="#Transform"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="Label">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasDoSection">
    <rdfs:domain rdf:resource="#Script"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasSpeedTypeSection">
    <rdfs:domain rdf:resource="#Script"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasRepeatSection">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#int"/>
    <rdfs:domain rdf:resource="#Script"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasSpeedSection">
    <rdfs:domain rdf:resource="#Script"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasBeforeSection">
    <rdfs:domain rdf:resource="#Script"/>
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:ID="hasVariablesSection">
    <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
    <rdfs:domain rdf:resource="#Script"/>
  </owl:DatatypeProperty>
  <owl:DatatypeProperty rdf:about="#direction">
    <rdfs:range>
      <owl:DataRange>
        <owl:oneOf rdf:parseType="Resource">
          <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >forward</rdf:first>
          <rdf:rest rdf:parseType="Resource">
            <rdf:rest rdf:parseType="Resource">
              <rdf:rest rdf:parseType="Resource">
                <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                >right</rdf:first>
                <rdf:rest rdf:parseType="Resource">
                  <rdf:rest rdf:parseType="Resource">
                    <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
                    <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                    >down</rdf:first>
                  </rdf:rest>
                  <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
                  >up</rdf:first>
                </rdf:rest>
              </rdf:rest>
              <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
              >left</rdf:first>
            </rdf:rest>
            <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >backward</rdf:first>
          </rdf:rest>
        </owl:oneOf>
      </owl:DataRange>
  </rdfs:range>
  <rdfs:domain rdf:resource="#DirectionDescription"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="hasConditionSection">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Script"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="#hasConstructMethod">
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf rdf:parseType="Resource">
        <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >Appear</rdf:first>
        <rdf:rest rdf:parseType="Resource">
          <rdf:rest rdf:parseType="Resource">
            <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >ZoomIn</rdf:first>
            <rdf:rest rdf:parseType="Resource">
              <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
              <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
              >FadeIn</rdf:first>
            </rdf:rest>
          </rdf:rest>
          <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >Grow</rdf:first>
        </rdf:rest>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
  <rdfs:domain rdf:resource="#Construct"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="#hasCondition">
  <rdfs:domain rdf:resource="#ConditionalOperator"/>
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="#hasDestructMethod">
  <rdfs:domain rdf:resource="#Destruct"/>
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf rdf:parseType="Resource">
        <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >Disappear</rdf:first>
        <rdf:rest rdf:parseType="Resource">
          <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >FadeOut</rdf:first>
          <rdf:rest rdf:parseType="Resource">
            <rdf:rest rdf:parseType="Resource">
              <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
              >Shrink</rdf:first>
              <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
            </rdf:rest>
            <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
            >ZoomOut</rdf:first>
          </rdf:rest>
        </rdf:rest>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:ID="hasPredicate">
  <rdfs:range rdf:resource="http://www.w3.org/2001/XMLSchema#string"/>
  <rdfs:domain rdf:resource="#Context"/>
</owl:DatatypeProperty>
<owl:DatatypeProperty rdf:about="#hasManner">
  <rdfs:range>
    <owl:DataRange>
      <owl:oneOf rdf:parseType="Resource">
        <rdf:rest rdf:parseType="Resource">
          <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
          >Smooth</rdf:first>
          <rdf:rest rdf:resource="http://www.w3.org/1999/02/22-rdf-syntax-ns#nil"/>
        </rdf:rest>
        <rdf:first rdf:datatype="http://www.w3.org/2001/XMLSchema#string"
        >AtOnce</rdf:first>
      </owl:oneOf>
    </owl:DataRange>
  </rdfs:range>
  <rdfs:domain>
    <owl:Class>
      <owl:unionOf rdf:parseType="Collection">
        <owl:Class rdf:about="#Group"/>
        <owl:Class rdf:about="#Ungroup"/>
        <owl:Class rdf:about="#Orientate"/>
        <owl:Class rdf:about="#Position"/>
        </owl:unionOf>
      </owl:Class>
    </rdfs:domain>
  </owl:DatatypeProperty>
  <owl:TransitiveProperty rdf:about="#subActorOf">
    <rdfs:range rdf:resource="#Actor"/>
    <rdf:type rdf:resource="http://www.w3.org/2002/07/owl#ObjectProperty"/>
    <owl:inverseOf rdf:resource="#superActorOf"/>
    <rdfs:domain rdf:resource="#Actor"/>
  </owl:TransitiveProperty>
</rdf:RDF>
```

APPENDIX B

Script Language BNF Specification

⟨*script*⟩ ::= ['/speed' ⟨*speed-declaration*⟩]
    ['/speedtype' ⟨*speedtype-declaration*⟩]
    ['/repeat' ⟨*repeat-declaration*⟩]
    ['/variables' ⟨*variables-declaration*⟩]
    ['/condition' ⟨*condition-declaration*⟩]
    ['/before' ⟨*block*⟩]
    ['/do' ⟨*block*⟩]
    ['/after' ⟨*block*⟩]

⟨*speed-declaration*⟩ ::= ⟨*velocity*⟩
    | ⟨*velocity*⟩ 'for' ⟨*id-list*⟩ { ';' ⟨*velocity*⟩ 'for' ⟨*id-list*⟩ }

⟨*velocity*⟩ ::= '`very slow`' | '`slow`' | '`normal`' | '`fast`' | '`very fast`'

⟨*speedtype-declaration*⟩ ::= ⟨*type*⟩
    | ⟨*type*⟩ 'for' ⟨*id-list*⟩ { ';' ⟨*type*⟩ 'for' ⟨*id-list*⟩ }

⟨*type*⟩ ::= '`decelerate`' | '`accelerate`' | '`constant`' | '`accelerate/decelerate`'

⟨*repeat-declaration*⟩ ::= (⟨*numeral*⟩ | ⟨*variable*⟩) 'time(s)'

⟨*variables-declaration*⟩ ::= ⟨*assignment*⟩ { ';' ⟨*assignment*⟩ }

⟨*condition-declaration*⟩ ::= ⟨*conditional-exp*⟩

⟨*conditional-exp*⟩ ::= ⟨*disjunctive-exp*⟩

⟨*block*⟩ ::= { ⟨*statement*⟩ }

```
⟨statement⟩ ::= ⟨assignment⟩
      | ⟨function-call⟩
      | ⟨returning-exp⟩
      | ’increment’ ⟨postfix-exp⟩ [ ’by’ ⟨exp⟩ ]
      | ’decrement’ ⟨postfix-exp⟩ [ ’by’ ⟨exp⟩ ]
      | ’while’ ⟨exp⟩ ’do’ ⟨block⟩ ’end’
      | ’when’ ⟨exp⟩ ’do’ ⟨block⟩ [ ’otherwise’ ⟨block⟩ ] ’end’
      | ’for’ ⟨identifier⟩ ’from’ ⟨exp⟩ ’to’ ⟨exp⟩ [ ’by’ ⟨exp⟩ ] ’do’ ⟨block⟩ ’end’

⟨exp⟩ ::= ⟨assignment-exp⟩

⟨assignment-exp⟩ ::= ⟨disjunctive-exp⟩ | ⟨assignment⟩

⟨assignment⟩ ::= ’assign’ ⟨exp⟩ ’to’ ⟨variable⟩

⟨disjunctive-exp⟩ ::= ⟨conjunctive-exp⟩
      | ⟨disjunctive-exp⟩ ’or’ ⟨conjunctive-exp⟩

⟨conjunctive-exp⟩ ::= ⟨equality-exp⟩
      | ⟨conjunctive-exp⟩ ’and’ ⟨equality-exp⟩

⟨equality-exp⟩ ::= ⟨relational-exp⟩
      | ⟨equality-exp⟩ ( ’=’ | ’<>’ ) ⟨relational-exp⟩

⟨relational-exp⟩ ::= ⟨additive-exp⟩
      | ⟨relational-exp⟩ ( ’<’ | ’>’ | ’<=’ | ’>=’ ) ⟨additive-exp⟩

⟨additive-exp⟩ ::= ⟨multiplicative-exp⟩
      | ⟨additive-exp⟩ ( ’+’ | ’-’) ⟨multiplicative-exp⟩

⟨multiplicative-exp⟩ ::= ⟨unary-exp⟩
      | ⟨multiplicative-exp⟩ ( ’*’ | ’/’ | ’%’ ) ⟨unary-exp⟩

⟨unary-exp⟩ ::= ’increment’ ⟨unary-exp⟩
      | ’decrement’ ⟨unary-exp⟩
      | ⟨postfix-exp⟩
      | ( ’+’ | ’-’ ) ⟨unary-exp⟩
      | ⟨negative-exp⟩

⟨negative-exp⟩ ::= ’not’ ⟨unary-exp⟩
      | ⟨postfix-exp⟩

⟨postfix-exp⟩ ::= ⟨primary⟩ | ⟨id-exp⟩

⟨primary⟩ ::= ⟨literal⟩ | ’(’ ⟨exp⟩ ’)’ | ⟨function-call⟩ | ⟨attribute-access⟩
```

⟨*attribute-access*⟩ ::= ’`ThisAction`’ ’.’ ⟨*identifier*⟩
  | ’`ThisEnvironment`’ ’.’ ⟨*identifier*⟩

⟨*function-call*⟩ ::= ’invoke’ ⟨*function-name*⟩ [’with’ ’(’ [⟨*explist*⟩] ’)’]

⟨*function-name*⟩ ::= ⟨*id-exp*⟩

⟨*explist*⟩ ::= { ⟨*exp*⟩ ’,’ } ⟨*exp*⟩

⟨*function*⟩ ::= ’define’ ⟨*function-name*⟩ [’with’ ’(’ [⟨*parlist*⟩] ’)’] ⟨*block*⟩
  ’end’

⟨*parlist*⟩ ::= ⟨*identifier*⟩ {’,’ ⟨*identifier*⟩ }

⟨*returning-exp*⟩ ::= ’return’ ⟨*exp*⟩

⟨*variable*⟩ ::= ⟨*identifier*⟩
  | ’`ThisAction`’ ’.’ ⟨*identifier*⟩

⟨*id-exp*⟩ ::= ⟨*identifier*⟩
  | ⟨*id-exp*⟩ ’.’ ⟨*identifier*⟩

⟨*identifier*⟩ ::= ⟨*character*⟩ { (⟨*character*⟩ | ⟨*digit*⟩) }

⟨*id-list*⟩ ::= ⟨*identifier*⟩ { ’,’ ⟨*identifier*⟩ }

⟨*literal*⟩ ::= ⟨*numeral*⟩ | ⟨*boolean*⟩ | ⟨*string*⟩ | ⟨*character*⟩

⟨*boolean*⟩ ::= ’`true`’ | ’`false`’

⟨*numeral*⟩ ::= [’`-`’] ⟨*number*⟩ { ⟨*digit*⟩ } [’.’ ⟨*digit*⟩ { ⟨*digit*⟩ }]
  | [’`-`’] ’`0`’ [’.’ ⟨*digit*⟩ { ⟨*digit*⟩ }]

⟨*string*⟩ ::= ’`"`’ { (⟨*character*⟩ | ⟨*digit*⟩) - ’`"`’ } ’`"`’

⟨*character*⟩ ::= ”’ (⟨*character*⟩ | ⟨*digit*⟩) - ”’ ”’

⟨*character*⟩ ::= ’`a`’ | ’`b`’ | ’`c`’ | ’`d`’ | ’`e`’ | ’`f`’ | ’`g`’ | ’`h`’ | ’`i`’ | ’`j`’ | ’`k`’ | ’`l`’ | ’`m`’ |
  ’`n`’ | ’`o`’ | ’`p`’ | ’`q`’ | ’`r`’ | ’`s`’ | ’`t`’ | ’`u`’ | ’`v`’ | ’`w`’ | ’`x`’ | ’`y`’ | ’`z`’ | ’`A`’ | ’`B`’ | ’`C`’
  | ’`D`’ | ’`E`’ | ’`F`’ | ’`G`’ | ’`H`’ | ’`I`’ | ’`J`’ | ’`K`’ | ’`L`’ | ’`M`’ | ’`N`’ | ’`O`’ | ’`P`’ | ’`Q`’ | ’`R`’ |
  ’`S`’ | ’`T`’ | ’`U`’ | ’`V`’ | ’`W`’ | ’`X`’ | ’`Y`’ | ’`Z`’

⟨*digit*⟩ ::= ’`0`’ | ⟨*number*⟩

⟨*number*⟩ ::= ’`1`’ | ’`2`’ | ’`3`’ | ’`4`’ | ’`5`’ | ’`6`’ | ’`7`’ | ’`8`’ | ’`9`’

APPENDIX C

Resources: Experiment

C.1 Introductory Document (in Dutch)

C.1.1 Modelleren van gedrag in Virtuele Omgevingen (gebruik makende van VR-WISE)

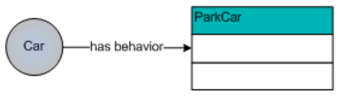
Virtuele Realiteit is een manier om het visualiseren, manipuleren en interageren met computers en complexe gegevens mogelijk te maken. Virtuele Omgevingen zijn computer-gegenereerde werelden die het effect creëren van een interactieve 3-dimensionele wereld waarin objecten kunnen voorgesteld en gemanipuleerd worden als dusdanig. Heel wat software applicaties zijn reeds beschikbaar om op een intuitieve manier Virtuele Omgevingen te bouwen maar deze leggen zich vooral toe op het maken van het visuele gedeelte. Het dynamische gedeelte (gedrag) moet in de praktijk veelal manueel geprogrammeerd worden. Daarom hebben we een benadering ontwikkeld met het doel om ook het dynamische gedeelte op een intuitieve manier te specificeren, namelijk door middel van een grafische notatie (diagrammen).

Deze benadering is onderverdeeld in twee stappen:

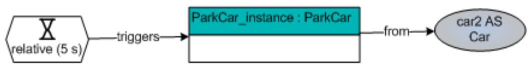
- **Behaviour Definition (Diagram)**. Hier gaan we het eigenlijke gedrag definiëren, maar op dit moment nog onafhankelijk van het object waaraan we het willen koppelen en onafhankelijk van de manier waarop het uiteindelijk opgeroepen wordt.
- **Behaviour Invocation (Diagram)**. Hier gaan we dan het gedrag dat we zojuist gedefinieerd hebben toekennen aan de objecten in de Virtuele Omgeving, en we gaan de events toevoegen die uiteindelijk ons gedrag gaan oproepen.

C.1.1.1 Voorbeeld

Een eerste voorbeeldje is een Behaviour Definition Diagram waarbij een actor (gerepresenteerd door de cirkel) genaamd ”Car” een gedrag aan wordt toegekend, namelijk ”ParkCar” (door middel van een has-behaviour link). Hiermee begin je een definitie van een gedrag binnen onze benadering. Vervolgens kan je binnen het ParkCar gedrag met behulp van primitieve acties (en deze met elkaar te linken) het uiteindelijke gedrag van deze Car definieren.

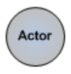
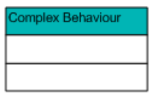


Het onderstaande diagram is van het overeenkomstige Behaviour Invocation Diagram waarbij een object in onze Virtuele Omgeving (instantie), namelijk ”car2”, gekoppeld is met de actor ”Car” (uit het gedrag dat in het Behaviour Definition Diagram aangemaakt werd). Hiermee kunnen we zeggen dat een object alle behaviours heeft die geassocieerd zijn met Car. De ParkCar_instance is dan een referentie naar het gedrag ParkCar, gedefinieerd in het Behaviour Definition Diagram. Ten slotte is er een event gekoppeld aan het gedrag dat ervoor zal zorgen dat het gedrag ook daadwerkelijk uitgevoerd wordt.



C.1.1.2 Definitie van het gedrag

Een eerste soort van diagram dat je dus nodig hebt om een gedrag te definiëren is een Behaviour Definition Diagram. In dit diagram kan je onder andere gebruik maken van de volgende grafische elementen:

Notatie	Uitleg
	**Actor**: Object waarvoor we een gedrag willen definiëren. - Naam
	**Composite Behaviour:** Complex gedrag, kan verschillende acties bevatten. - Naam

Vervolgens zijn er een hele reeks van primitieve acties beschikbaar die gebruikt kunnen worden in de definities van het gedrag zoals aangegeven in tabellen C.1 en C.3. De verschillende acties die gebruikt werden kunnen aan elkaar gelinkt worden door middel van de operatoren. De meest voorkomende set van operatoren zijn de temporal operators (zie figuur C.1) en de lifetime operators zoals aangegeven in tabel C.2.

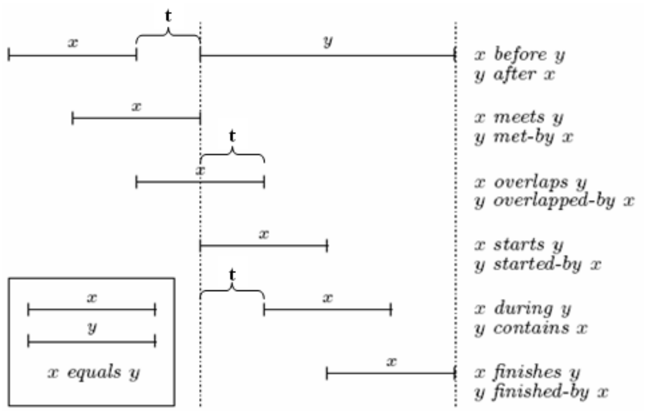


Figure C.1: *Temporal operators*

C.1.1.3 Invocatie van het gedrag

Nadat je een Behaviour Definition Diagram hebt gemaakt voor het gedrag, heb je een Behaviour Invocation Diagram nodig om dit gedrag te koppelen aan de objecten, en om de events toe te voegen om het gedrag te triggeren. Hiervoor kan je gebruik maken van de grafische elementen zoals aangegeven in tabel C.4.

C.1.2 Grondplan

Figuur C.2 geeft een overzichtje (van bovenaf gezien) van de Virtuele Omgeving. Deze Virtuele Omgeving zal gebruikt worden tijdens het experiment. De namen van de verschillende gebouwen staan in de legende onderaan de figuur.

C.2 Assignments (in Dutch)

Maak de volgende oefeningen zoals aangegeven in de opgaves hieronder en in de video's.

- Oefening 1 : BusManeuvre
  Voer een manoeuvre uit met de bus, genaamd "bus1", als volgt. De bus maakt een ommekeer door op het volgende kruispunt links af te slaan en daarna achterwaarts terug te keren zoals weergegeven op de video en vervolgens verder te gaan op dezelfde weg. De bus zal uiteindelijk stoppen aan de bushalte.

- Oefening 2 : CityEvolution
  Laat het gebouw "fortisbuilding" afbreken. Terwijl deze afbraak nog bezig is wordt het "electrabelbuilding" volledig heropgebouwd (lees: verandert in een andere vorm), namelijk in een "HighestBuilding". Enige tijd nadat dit gebeurd is zal op de oude plaats van het fortisbuilding, dus naast het "hiltonbuilding", een nieuw gebouw worden neergeplant, namelijk "bramspalace".

C.3 Questionnaire (in Dutch)

Deelnemer # _______

Probeer onderstaande vragen naar best vermogen te beantwoorden.

1. Hoe vaak maak je gebruik van computers? (nooit, maandelijks, wekelijks, dagelijks)
2. Heb je al ooit gebruik gemaakt van een VR systeem? (bv. Videogames, 3D omgevingen,. . .)
3. Had je al enige kennis van grafische notaties (zoals UML, ORM,. . .)?
4. Had je al enige voorkennis van Microsoft Visio?
5. Welk gedrag vond je het moeilijkste om te modelleren?
6. Welk aspect van de gedragingen vond je het moeilijkst en waarom?
7. Waren de symbolen (icoontjes) gebruikt in de grafische notatie intuïtief genoeg om er uit af te leiden wat het gedrag voorstelt?
8. Wat vond je niet intuïtief, of wat kan er verbeterd worden?
   - De grafische notatie zelf: bepaalde primitieve acties, de connecties tussen de verschillende acties.
   - De benadering met de twee stappen: Behaviour Definition en Behaviour Invocation.
   - De software applicaties.

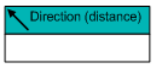
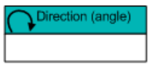
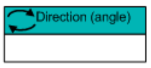
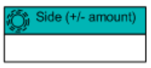
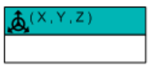
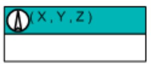
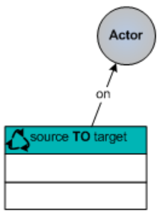
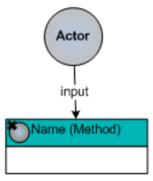
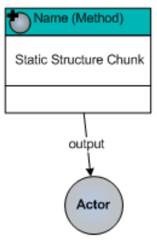
Notatie	Uitleg
	**Move:** Bewegen in een bepaalde richting over een bepaalde afstand. - Richting: forward, backward, left, right, up, down,. . . - Afstand
	**Turn:** Draaien in een bepaalde richting over een bepaalde hoek. - Richting: left, right - Hoek
	**Roll:** Rollen in een bepaalde richting over een bepaalde hoek. - Richting: forward, backward, left, right - Hoek
	**Resize:** Uitrekken (inkrimpen) vanuit de zijde(s) over een bepaalde hoeveelheid. - Zijdes: front, back, left, right, top, bottom - Type (grow of shrink) en een hoeveelheid
	**Position:** Positioneren op een specifieke locatie. - x, y, z coördinaat
	**Orient:** Oriënteren in een specifieke richting. - x, y, z draaingshoeken over de respectievelijke assen
	**Transform:** Transformeren van de vorm van een object naar een andere vorm. - Oude vorm (uit een lijst van voorgedefinieerde vormen) - Nieuwe vorm (uit een lijst van voorgedefinieerde vormen)
	**Destruct:** Verwijderen van een object uit de scene. - Manier: disappear, shrink, zoom-out - Input actor (het te verwijderen object)
	**Construct:** Toevoegen van een object aan de scene. - Manier: appear, grow, zoom-in - Output actor (het toe te voegen object)

Table C.1: *Grafische Elementen: Behaviour Definition Diagram*

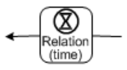
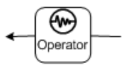
Notatie	Uitleg
	**Temporal Operator:** Koppel acties aan elkaar in functie van de tijd. - Relatie: before, after, meets, during, overlaps,... - Tijd
	**Lifetime Operator:** Een relatie die de levensduur van een actie controleert. - Operator: enable, disable, suspend, resume

Table C.2: *Grafische Elementen: Behaviour Definition Diagram (con't)*

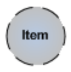
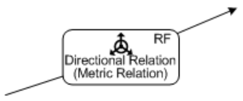
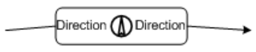
Notatie	Uitleg
	**Item**: Een object dat we willen plaatsen of waarnaar we willen refereren om een ander object te plaatsen. - Naam
	**Spatial Relation:** Een relatie die een object plaatst relatief t.o.v. een ander object. - Richting: left-of, right-of, above, below, in-front-of, behind - Afstand
	**Orientation Relations:** Een relatie die een object heroriënteert t.o.v. een ander object. - Zijde van oorsprong: front, back, left, right, top, bottom - Zijde van bestemming: front, back, left, right, top, bottom

Table C.3: *Grafische Elementen: Structure Chunk*

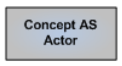
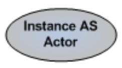
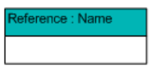
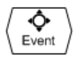
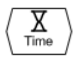
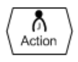
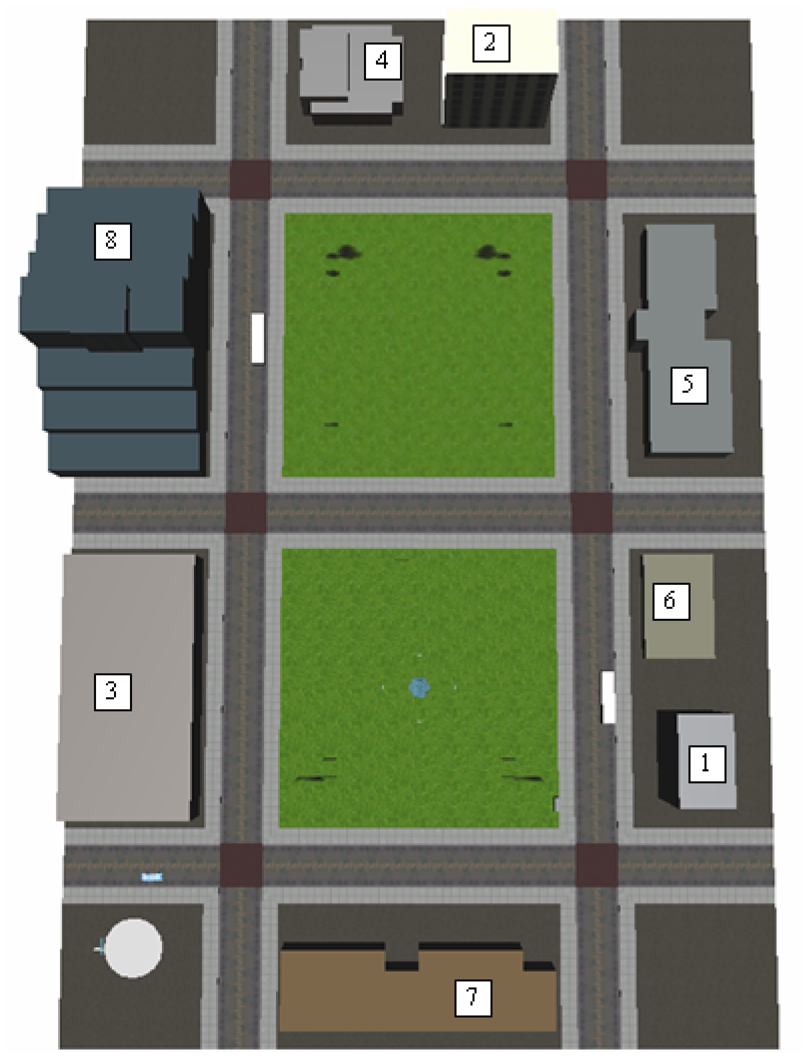
Notatie	Uitleg
	**Concept**: Een object-type van de Virtuele Wereld, moet toegekend worden aan een actor. - Naam - Naam van de actor
	**Instance:** Een werkelijk object in de Virtuele Wereld, moet toegekend worden aan een actor. - Naam - Naam van de actor
	**Behaviour Reference:** Referentie naar een gedrag dat gedefinieerd werd in een Behaviour Definition Diagram. - Naam van het gedrag waarnaar het refereert - Naam
	**Initialize event:** Een trigger op basis van een conditie die geldig wordt op een bepaald tijdstip. - Conditie waaraan voldaan moet worden om het gedrag op te roepen
	**Time event:** Een trigger op basis van het verstrijken van de tijd. - Absolute tijd of Relatieve tijd
	**User event:** Een trigger op basis van een actie van de gebruiker. - Type: OnKeyPress, OnTouch, OnProxy, OnSelect - Eventuele argumenten

Table C.4: *Grafische Elementen: Behaviour Invocation Diagram*



1. adminbuilding
2. hiltonbuilding
3. dexiabuilding
4. fortisbuilding
5. elactrabelbuilding
6. policestationbuilding
7. cityhallbuilding
8. btbuilding

Overige objecten:
bus1, bus2
car2
chopper
fountain

Figure C.2: *Stadsplan*

References

C. Alexander, S. Ishikawa, and M. Silverstein. *A Pattern Language: Towns, Buildings, Construction.* Oxford University Press, 1977. 153

J. F. Allen. Time and time again: The many ways to represent time. *International Journal of Intelligent Systems*, 6(4):341–355, 1991. 121

Y. Arafa, K. Kamyab, S. Kshirsagar, N. Magnenat-Thalmann, A. Guye-Vuillaume, and D. Thalmann. Two approaches to scripting character animation. In *Proceedings AAMAS-02 Workshop on Embodied Conversational Agents*, Bologna, Italy, 2002. 51

S. Arjomandy and T. J. Smedley. Visual specification of behaviours in vrml worlds. In *Proceedings of the ninth international conference on 3D Web technology*, pages 127–133, California, USA, 2004. ACM Press. 41

J. W. Backus, J. H. Wegstein, A. van Wijngaarden, M. Woodger, F. L. Bauer, J. Green, C. Katz, J. McCarthy, A. J. Perlis, H. Rutishauser, K. Samelson, and B. Vauquois. Revised report on the algorithmic language algol 60. *Communications of the ACM*, 3(5):299–314, 1960. 140

N. I. Badler, R. Bindiganavale, J. Allbeck, W. Schuler, L. Zhao, S.-J. Lee, H. Shin, and M. Palmer. Parameterized action representation and natural language instructions for dynamic behavior modification of embodied agents. In *Proceedings of the AAAI Spring Symposium 2000*, Stanford, California, 2000. 47

N. I. Badler, R. Bindiganavale, J. Allbeck, W. Schuler, L. Zhao, and M. Palmer. Parameterized action representation for virtual human agents. In *Proceedings of Embodied conversational agents*, pages 256–284. MIT Press, 2001. 47

F. Bernier, E. Boivin, D. Laurendeau, M. Mokhtari, and F. Lemieux. Conceptual models for describing virtual worlds. In *WSCG Posters Proceedings*, pages 25–28, Plezn, Czech Republic, 2004. UNION Agency - Science Press. 59

W. Bille. *Conceptual Modeling of Complex Objects for Virtual Environments - A Formal Approach*. PhD thesis, Vrije Universiteit Brussel, 2007. 95, 101

W. Bille, O. De Troyer, F. Kleinermann, B. Pellens, and R. Romero. Using ontologies to build virtual worlds for the web. In P. Isaias and N. Karmakar, editors, *Proceedings of the IADIS International Conference WWW/Internet 2004 (ICWI2004)*, volume 1, pages 683–690, Madrid, Spain, 2004a. IADIS PRESS. 62

W. Bille, B. Pellens, F. Kleinermann, and O. De Troyer. Intelligent modelling of virtual worlds using domain ontologies. In L. Sheremetov and M. Alvarado, editors, *Proceedings of MICAI 2004 Intelligent Computing Workshops*, pages 272–279, Mexico City, Mexico, 2004b. 62

W. Bille, O. De Troyer, B. Pellens, and F. Kleinermann. Conceptual modeling of articulated bodies in virtual environments. In H. Thwaites, editor, *Proceedings of the 11th International Conference on Virtual Systems and Multimedia*, pages 17–26, Ghent, Belgium, 2005. Archaeolingua. 65

S. Bjork and J. Holopainen. *Patterns in Game Design*. Game Development Series. Charles River Media, 1st edition, December 2004. 52

L. Blackwell, B. von Konsky, and M. Robey. Petri net script: a visual language for describing action, behavior and plot. In *Proceedings of the 24th Australasian Computer Science Conference (ACSC 2001)*, volume 11 of *ACM International Conference Proceedings Series*, pages 29–37, Queensland, Australia, 2001. IEEE Computer Society. 39

D. Bowman. Formalizing the design, evaluation, and application of interaction techniques for immersive virtual environments. *Journal of Visual Languages and Computing*, 10(1):37–53, 1999. 7

E. J. Braude. *Software Engineering: An Object-Oriented Perspective*. Wiley, 1st edition, 2000. 58

G. C. Burdea and P. Coiffet. *Virtual Reality Technology*. Wiley-Interscience, 2nd edition, 2003. 184

T. Burrows. *The Specification of Behaviour in Virtual Environments*. PhD thesis, Liverpool John Moores University, Liverpool, UK, June 2004. 46

T. Burrows and D. England. Yable: Yet another behaviour language. In *Proceedings of the tenth international conference on 3D Web technology*, pages 65–73, Bangor, United Kingdom, 2005. ACM Press. 46

J. Busby, Z. Parrish, and J. VanEenwyk. *Mastering Unreal Technology: The Art of Level Design*. Sams, December 2004. 7

M. Cavazza, S. Hartley, J.-L. Lugrin, and M. Le Bras. Qualitative physics in virtual environments. In *Proceedings of the 9th international conference on Intelligent user interface*, pages 54–61, Funchal, Portugal, 2004a. ACM Press. 52

M. Cavazza, S. Hartley, J.-L. Lugrin, P. Libardi, and M. Le Bras. New behavioural approaches for virtual environments. In *Proceedings of the International Conference on Entertainment Computing*, volume 3166 of *Lecture Notes in Computer Science*, pages 23–31, Eindhoven, Holland, 2004b. Springer-Verlag. 52

E. Cerezo, A. Pina, and F. Seron. Motion and behaviour modelling: state of art and new trends. *The Visual Computer*, 15:124–146, 1999. 53

S. Clarke-Wilson. *Digital illusion: entertaining the future with high technology*, chapter 13, pages 229–239. ACM Press/Addison-Wesley Publishing Co, 1998. 51

K. Coninx, O. De Troyer, C. Raymaekers, and F. Kleinermann. Vr-demo: a tool-supported approach facilitating flexible development of virtual environments using conceptual modelling. In D. Coutellier and X. Fischer, editors, *Proceedings of Virtual Concept 2006*, Cancun, Mexico, 2006. Springer-Verlag. 202

M. Conway. *Alice: Easy-to-Learn 3D Scripting for Novices*. PhD thesis, University of Virginia, Virginia, USA, December 1997. 43

M. Conway, S. Audia, T. Burnette, D. Cosgrove, and K. Christiansen. Alice: Lessons learned from building a 3d system for novices. In *Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 486–493, The Hague, The Netherlands, 2000. ACM Press. 43

R. Dachselt. Contigra: Towards a document-based approach to 3d components. In *Proceedings of the Structured Design of Virtual Environments and 3D-Components Workshop*, Paderborn, Germany, 2001. 31

R. Dachselt and E. Rukzio. Behavior3d: an xml-based framework for 3d graphics behavior. In C. Bouville, editor, *Proceedings of the eighth international conference on 3D Web technology*, pages 101–112, Saint Malo, France, 2003. ACM Press. 32

R. Dachselt, M. Hinz, and K. Meisner. Contigra: an xml-based architecture for component-oriented 3d applications. In *Proceedings of the seventh international conference on 3D Web technology*, pages 155–163, Arizona, USA, 2002. ACM Press. 31

J. De Boeck. *A User and Designer Perspective on Multimodal Interaction in 3D Environments*. PhD thesis, Universiteit Hasselt, January 2007. 226

O. De Troyer, W. Bille, R. Romero, and P. Stuer. On generating virtual worlds from domain ontologies. In T.-S. Chua and T. L. Kunii, editors, *Proceedings of the 9th International Conference on Multi-Media Modeling*, pages 279–294, Taipei, Taiwan, 2003. Tamkang University. 61

O. De Troyer, F. Kleinermann, H. Mansouri, B. Pellens, W. Bille, and V. Fomenko. Developing semantic vr-shops for e-commerce. *Special Issue of VIRTUAL REALITY: Virtual Reality in the e-Society*, 1359-4338, 2006. 226

F. Devillers and S. Donikian. A scenario language to orchestrate virtual world evolution. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 265–275, San Diego, California, 2003. Eurographics Association. 48

F. Devillers, S. Donikian, F. Lamarche, and J. F. Taille. A programming environment for behavioral animation. *Journal of Visualization and Computer Animation*, 13:263–274, 2002. 48

A. Diaz and A. Fernandez. A pattern language for virtual environments. *Journal of Network and Computer Applications*, 23(3):291–309, 2000. 51

T. Dillon and P. L. Tan. *Object-Oriented Conceptual Modeling*, chapter 4, pages 87–123. Prentice Hall, 1993. 58

S. Donikian. Hpts: a behaviour modelling language for autonomous agents. In *Proceedings of the fifth international conference on Autonomous agents*, pages 401–408, Montreal, Canada, 2001. ACM Press. 48

Ecma, editor. *Ecmascript Language Specification (Open Documents Standards Library)*. Iuniverse Inc, 3rd edition, April 2000. 15

M.J. Egenhofer. Topological relations in 3d. Technical report, University of Maine, USA, 1995. 65

C. Fencott. Towards a design methodology for virtual environments. In *Proceedings of the Workshop on User Centered Design and Implementation of Virtual Environments*, York, UK, 1999. 51

P. A. Fishwick. 3d behavioral model design for simulation and software engineering. In *Proceedings of the fifth symposium on Virtual reality modeling language (Web3D-VRML)*, pages 7–16, California, USA, 2000. ACM Press. 42

E. Folmer. Usability patterns in games. In *Proceedings of the Futureplay 2006 conference*, Ontario, Canada, 2006. 52

A. U. Frank. Formal models for cognition, taxonomy of spatial location description and frames of reference. In C. Freksa, C. Habel, and K. F. Wender, editors, *Spatial Cognition, An Interdisciplinary Approach to Representing and Processing Spatial Knowledge*, volume 1404 of *Lecture Notes in Computer Science*, pages 293–312. Springer-Verlag, 1998. 91

Andrew U. Frank. Qualitative spatial reasoning: Cardinal directions as an example. *International Journal of Geographical Information Science*, 10 (3):269290, 1996. 64

D. Fu, R. Houlette, and R. Jensen. A visual environment for rapid behavior definition. In *Proceedings of the 2003 Conference on Behavior Representation in Modeling and Simulation*, Arizona, USA, 2003. 41

E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional, 1995. 153

K.P. Gapp. Basic meanings of spatial relations: Computation and evaluation in 3d space. In *In proceedings of the National Conference on Artificial Intelligence*, pages 1393–1398, Seattle, USA, 1994. 64

C. Geiger, V. Paelke, C. Reimann, and W. Rosenbach. A framework for the structured design of vr/ar content. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 75–82, Seoul, Korea, 2000. ACM Press. 39

C. Geiger, V. Paelke, C. Reimann, and W. Rosenbach. Structured design of interactive virtual and augmented reality content. In *Proceedings of the Structured Design of Virtual Environments and 3D-Components Workshop*, Paderborn, Germany, 2001. 37

P. Gerstl and S. Pribbenow. A conceptual theory of part-whole relations and its applications. *Data and Knowledge Engineering*, 20(3):305–322, 1996. 65

T. R. Gruber. A translation approach to portable ontology specifications. *Special Issue of Journal of Knowledge Acquisition: Current issues in knowledge modeling*, 5(2):199–220, 1993. 60

B. Grutzmacher, R. Wages, and G. Trogemann. An authoring system for non-linear vr scenarios. In *Proceedings of the Ninth International Conference on Virtual Systems and Multimedia*, pages 634–641, Montreal, Canada, 2003. 226

N. Guarino and P. Giaretta. Ontologies and knowledge bases: Towards a terminological clarification. In N. J. I. Mars, editor, *Proceedings of 2nd Internationl Conference on Building and Sharing Very Large-Scale Knowledge Bases*, pages 25–32, Amsterdam, The Netherlands, 1995. IOS Press. 60

T. Halpin. *Information Modeling and Relational Databases, From Conceptual Analysis to Logical Design*. Morgan Kaufmann, 2001. 59

A. Herskovits. *Language and Spatial Cognition: An Interdisciplinary Study of the Prepositions in English*. Cambridge University Press, 1987. 98

N. Hirzalla, B. Falchuk, and Ahmed Karmouch. A temporal model for interactive multimedia systems. *IEEE Multimedia*, 2(3):24–31, 1995. 121

J. R. Hobbs and F. Pan. An ontology of time for the semantic web. *ACM Transactions on Asian Language Processing (TALIP): Special issue on Temporal Information Processing*, 3(1):66–85, 2004. 84

J. R. Hobbs and J. Pustejovsky. Annotating and reasoning about time and events. In *Proceedings of the AAAI Spring Symposium on Logical Formulization of Commonsense Reasoning*, California, USA, 2003. Stanford University. 85

J. F. Hopkins and P. A. Fishwick. The rube framework for software modeling and customized 3-d visualization. *Journal of Visual Languages and Computing*, 14(1):97–117, 2003. 42

R. Houlette, D. Fu, and D. Ross. Towards an ai behavior toolkit for games. In *AAAI 2001 Spring Symposium on AI and Interactive Entertainment*, California, USA, 2001. 41

Z. Huang, A. Eliens, and C. Visser. Implementation of a scripting language for vrml/x3d-based embodied agents. In *Proceedings of the eighth international conference on 3D Web technology*, pages 91–100, Saint-Malo, France, 2003a. ACM Press. 44

Z. Huang, A. Eliens, and C. Visser. Xstep: a markup language for embodied agents. In *Proceedings of the 16th International Conference on Computer Animation and Social Agents*, page 105, California, USA, 2003b. IEEE Computer Society. 45

C. E. Hughes, M. J. Moshell, and D. Reed. *Handbook of Virtual Environments*, chapter 16, pages 333–352. Lawrence Erlbaum Associates, 2002. 4

R. Ierusalimschy. *Programming in Lua*. Lua.Org, 1 edition, 2003. 108

M. Iris, B. Lutowitz, and M. Evens. *Relational models of the lexicon*, chapter Problems of the part-whole relation, pages 261–288. Cambridge University Press, 1988. 65

R. J. K. Jacob, L. Deligiannidis, and S. Morrison. A software model and specification language for non-wimp user interfaces. *ACM Transactions on Computer-Human Interaction*, 6(1):1–46, 1999. 26

K. Jensen and G. Rozenberg, editors. *High-level Petri nets: theory and application*. Springer-Verlag, 1991. 40

M. Kallmann. *Object Interaction in Real-Time Virtual Environments*. PhD thesis, Swiss Federal Institute of Technology - EPFL, Lausanne, Swiss, January 2001. 45

M. Kallmann and D. Thalmann. Modeling behaviors of interactive objects for real-time virtual environments. *Journal of Visual Languages & Computing*, 13(2):177–195, 2002. 45

K. C. Kang, G. J. Kim, J. Y. Lee, and H. J. Kim. Prototype = function + behavior + form. *ACM SIGSOFT Software Engineering Notes*, 23(4): 44–49, 1998. 28

K. Kaur. *Designing Virtual Environments for Usability*. PhD thesis, City University, London, UK, June 1998. 51

G. D. Kessler. *Handbook of Virtual Environments*, chapter 12, pages 255–276. Lawrence Erlbaum Associates, 2002. 5

G. Kim, K. Kang, H. Kim, and J. Lee. Software engineering of virtual worlds. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 131–138, Taipei, Taiwan, 1998. ACM Press. 28

T. Kim and P. A. Fishwick. A 3d xml-based customized framework for dynamic models. In *Proceedings of the seventh international conference on 3D Web technology*, pages 103–109, Arizona, USA, 2002. ACM Press. 42

K. Kipper and M. Palmer. Representation of actions as an interlingua. In *NAACL-ANLP 2000 Workshop on Applied interlinguas: practical applications of interlingual approaches to NLP*, pages 12–17, Seattle, Washington, 2000. Association for Computational Linguistics. 47

F. Kleinermann, O. De Troyer, H. Mansouri, R. Romero, B. Pellens, and W. Bille. Designing semantic virtual reality applications. In *Proceedings of the 2nd INTUITION International Workshop*, Senlis, France, 2005. 61

S. MacDonald, D. Szafron, J. Schaeffer, J. Anvik, S. Bromling, and K. Tan. Generative design patterns. In *Proceedings of the 17th IEEE International Conference on Automated software engineering*, page 23, Edinburgh, Scotland, 2002. 154

N. Magnenat-Thalmann and D. Thalmann. Virtual reality software and technology. *Encyclopedia of Computer Science and Technology*, 41:331–361, 2000. Springer-Verlag. 3

H. Mansouri. Using semantic descriptions for building and querying virtual environments. Master's thesis, Vrije Universiteit Brussel, 2005. 195

C. McCarthy and D. Callele. *Virtools Dev User Guide*. Virtools SA, Paris, France, 2001. 49

D. L. McGuinness. Ontologies come of age. *Spinning the Semantic Web: Bringing the World Wide Web to Its Full Potential*, 1(1):171–188, 2003. 61

M. McNaughton, J. Redford, J. Schaeffer, D. Szafron, Y. Xiang, and B. Chaib-draa. Pattern-based ai scripting using scriptease. In *Proceedings of AI 2003 : advances in artificial intelligence*, volume 2671 of *Lecture notes in computer science*, pages 35–49, Halifax, Canada, 2003. Springer Verlag. 48

M. McNaughton, M. Cutumisu, D. Szafron, J. Schaeffer, J. Redford, and D. Parker. Scriptease: Generating scripting code for computer role-playing games. In *Proceedings of 19th IEEE International Conference on Automated Software Engineering*, pages 386–387, Linz, Austria, 2004a. IEEE Computer Society. 152

M. McNaughton, J. Schaeffer, D. Szafron, D. Parker, and J. Redford. Code generation for ai scripting in computer role-playing games. In *Proceedings of Game AI Workshop at AAAI-04*, pages 129–133, San Jose, USA, 2004b. 48

B. Messing and C. Hellmich. Using aspect oriented methods to add behaviour to x3d documents. In *Proceedings of the 11th International Conference on 3D Web Technology*, pages 97–107, Columbia, USA, 2006. ACM Press. 51

R. J. Millar, J. R. P Hanna, and S. M. Kealy. A review of behavioural animation. *Computers & Graphics*, 23:127–143, 1999. 53

J. P. Molina, A. S. Garcia, V. Lopez-Jaquero, and P. Gonzalez. Developing vr applications: the tres-d methodology. In *Proceedings of the 1st International Workshop On Methods and Tools for Designing VR Applications*, Ghent, Belgium, 2005. 51

K. L. Murdock. *3ds max 5 Bible*. Wiley Publishing, November 2002. 50

D. Nahon. Virtools and virtual reality. In *Proceedings of the 2nd International Intuition 2005 Workshop*, Senlis, France, 2005. 49

A. Nathan. *Windows Presentation Foundation Unleashed*. Sams, 1st edition, 2006. 163

J.J. Odell. Six different kinds of composition. *International Journal of Object Oriented Programming*, 5(8):10–15, 1994. 65

M. Oliveira, J. Crowcroft, and M. Slater. An innovative design approach to build virtual environment systems. In *Proceedings of the workshop on Virtual environments 2003*, volume 39 of *ACM International Conference Proceedings Series*, pages 143–151, Zurich, Switzerland, 2003. ACM Press. 51

B. Pellens, F. Kleinermann, O. De Troyer, and W. Bille. Overview of existing virtual reality modelling concepts. IWT SBO VR-DeMo (IWT 030248), Deliverable 1.1, Vrije Universiteit Brussel, Brussels, Belgium, 2004. 59

B. Pellens, W. Bille, O. De Troyer, and F. Kleinermann. Vr-wise: A conceptual modelling approach for virtual environments. In *Proceedings of the 1st International Methods and Tools for Virtual Reality (MeTo-VR 2005) workshop*, Ghent, Belgium, 2005a. 62

B. Pellens, O. De Troyer, W. Bille, and F. Kleinermann. Conceptual modeling of object behavior in a virtual environment. In X. Fischer and D. Coutellier, editors, *Research in Interactive Design, Proceedings of International Conference Virtual Concept*, pages 93–94, Biarritz, France, 2005b. Springer-Verlag. 106

B. Pellens, O. De Troyer, W. Bille, F. Kleinermann, and R. Romero. An ontology-driven approach for modeling behavior in virtual environments. In R. Meersman, Z. Tari, and P. Herrero, editors, *Proceedings of On the Move to Meaningful Internet Systems 2005: Ontology Mining and Engineering and its Use for Virtual Reality (WOMEUVR 2005) Workshop*, number 3762 in Lecture Notes in Computer Science, pages 1215–1224, Agia Napa, Cyprus, 2005c. Springer-Verlag. 81

B. Pellens, F. Kleinermann, O. De Troyer, J. De Boeck, E. Cuppens, T. De Weyer, and K. Coninx. State of the art in designing virtual reality applications. Iwt sbo vr-demo (iwt 030248), state-of-the-art report, Vrije Universiteit Brussel, Universiteit Hasselt, Belgium, 2006a. 53, 152

B. Pellens, F. Kleinermann, and O. De Troyer. Intuitively specifying object dynamics in virtual environments using vr-wise. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 334–337, Limassol, Cyprus, 2006b. ACM Press. 139

B. Pellens, F. Kleinermann, O. De Troyer, and W. Bille. Model-based design of virtual environment behavior. In H. Zha, Z. Pan, and H. Thwaites, editors, *Proceedings of the 12th International Conference on Virtual Systems and Multimedia*, number 4270 in Lecture Notes in Computer Science, pages 29–39, Xian, China, 2006c. Springer-Verlag. 106

B. Pellens, O. De Troyer, F. Kleinermann, and W. Bille. Conceptual modeling of behavior in a virtual environment. *Special Issue of International Journal of Product and Development*, 4(6):626–645, 2007. Inderscience Enterprises. 77

K. Perlin and A. Goldberg. Improv: a system for scripting interactive actors in virtual worlds. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 205–216, Los Angeles, USA, 1996. ACM Press. 43

R. Price, B. Srinivasan, and K. Ramamohanarao. Extending the unified modeling language to support spatiotemporal applications. In *Technology of Object-Oriented Languages and Systems*, pages 163–174, Melbourne, Australia, 1999. 81

E. Reiter and R. Dale. Building applied natural language generation systems. *Natural Language Engineering*, 3(1):57–87, 1997. 192

C. W. Reynolds. Flocks, herds, and schools: a distributed behavioral model. *Computer Graphics*, 21(4):25–34, 1987. 175

B. Roehl. Some thoughts on behavior in vr systems. http://www.ece.uwaterloo.ca/ broehl/behav.html, August 1995. 6

T. Roosendaal and S. Selleri. *The Official Blender 2.3 Guide: Free 3D Creation Suite for Modeling, Animation, and Rendering*. No Starch Press, April 2005. 50

S. Russel and P. Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 1995. 197

M.-I. Sanchez-Segura, A. de Antonio, and A. de Amescua. Interaction patterns for future interactive systems components. *Interacting with Computers*, 16:331–350, 2004. 35

M.-I. Sanchez-Segura, A. de Antonio, and A. de Amescua. Senda: A whole process to develop virtual environments. In M.-I. Sanchez-Segura, editor, *Developing Future Interactive Systems*, pages 92–114. Idea Group Publishing, 2005. 33

S. Sauer and G. Engels. Uml-based behavior specification of interactive multimedia applications. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 248–255, Stresa, Italy, 2001. 81

D. C. Schmidt. Using design patterns to develop reusable object-oriented communication software. *Communications of the ACM*, 38(10):65–74, 1995. 153

D. Selman. *Java 3D Programming*. Manning Publications, February 2002. 11, 196

J. Seo and G. J. Kim. A structured approach to virtual reality system design. *Presence*, 11(4):378–403, 2002. 30

S. Smith and D. Duke. Virtual environments as hybrid systems. In N. Dodgson and M. Austen, editors, *Proceedings of the 17th Eurographics Annual Conference*, pages 113–128, Cambridge, UK, 1999. Eurographics. 40

S. P. Smith, D. J. Duke, and J. S. Willans. Designing world objects for usable virtual environments. In P. Palanque and F. Paterno, editors, *Proceedings of the Workshop on Design, Specification and Verification of Interactive Systems: DSV-IS 2000*, pages 309–319, Limerick, Ireland, 2000. 40

F. Southey. Ossa: A modelling system for virtual realities based on conceptual graphs and production systems. Master's thesis, University of Guelph, September 1998. 35

F. Southey and J. G. Linders. Ossa : A conceptual modelling system for virtual realities. In H. S. Delugach and G. Stumme, editors, *Proceedings of the 9th International Conference on Conceptual Structures*, volume 2120 of *Lecture Notes in Computer Science*, pages 333–345, California, USA, 2001. Springer-Verlag. 35

S. Spaccapietra, C. Parent, and E. Zimanyi. Modeling time from a conceptual perspective. In *Proceedings of the seventh international conference on Information and knowledge management*, pages 432 – 440, Bethesda, USA, 1998. ACM Press. 121

R. Stuart. *Design of Virtual Environments*. Barricade Books, August 2001. 5

I. Sutherland. The ultimate display. In *Proceedings of the International Federation of Information Processing Congress*, volume 2, pages 506–509, 1965. 1

V. Tanriverdi and R. J. K. Jacob. Vrid: a design model and methodology for developing virtual reality interfaces. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 175–182, Alberta, Canada, 2001. ACM Press. 24

S. C. L. Terra and R. A. Metoyer. Performance timing for keyframe animation. In R. Boulic and D. K. Pai, editors, *Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 253 – 258, Grenoble, France, 2004. ACM Press. 14

B. Tversky. *Handbook of Memory*, chapter Remembering spaces, pages 363–378. New York: Oxford University Press, 2000. 97

M. Uschold and R. Jasper. A framework for understanding and classifying ontology applications. In *Proceedings of the IJCAI99 Workshop on Ontologies and Problem-Solving Methods(KRR5)*, Stockholm, Sweden, 1999. 61

A. Valente and J. A. Breuker. Towards principled core ontologies. In B. R. Gaines and M. Musen, editors, *Proceedings of the 10th Knowledge Acquisition for Knowledge-Based Systems Workshop*, pages 301–320, Banff, Canada, 1996. 64

A.C. Varzi. Parts, wholes, and part-whole relations: the prospects of mereotopology. *Journal of Data & Knowledge Engineering*, 20(3):259 – 286, 1996. 65

J. Vince. *Introduction to Virtual Reality*. Springer-Verlag, 1st edition, February 2004. 2

K. Walczak. Beh-vr: Modeling behavior of dynamic virtual reality contents. In H. Zha, Z. Pan, H. Thwaites, A. Addison, and M. Forte, editors, *Interactive Technologies and Sociotechnical Systems*, number 4270 in Lecture Notes in Computer Science, pages 40–51, Xian, China, 2006. Springer-Verlag. 51

A. Walsh and M. I. Bourges-Sevenier. *Core Web3D*. Core Series. Prentice Hall, September 2000. 11

K. N. Whitley. Visual programming languages and the empirical evidence for and against. *Journal of Visual Languages and Computing*, 8(1):109–142, 1997. 152

G. Wideman. *Visio 2003 Developer's Survival Pack.* Trafford Publishing, September 2003. 190

J. Willans. *Integrating behavioural design into the virtual environment development process.* PhD thesis, University of York, York, UK, November 2001. 40

P. Willemsen. *Behavior and Scenario Modeling for Real-Time Virtual Environments.* PhD thesis, University of Iowa, May 2000. 226

J. R. Wilson, R. M. Eastgate, and M. D'Cruz. *Handbook of Virtual Environments*, chapter 17, pages 353–378. Lawrence Erlbaum Associates, 2002. 26

M. E. Winston, R. Chaffin, and D. Herrmann. A taxonomy of part-whole relations. *Cognitive Science*, 11(4):417–444, 1987. 65

S. M. Yacoub and H. H. Ammar. *Pattern-Oriented Analysis and Design: Composing Patterns to Design Software Systems*, chapter 3, pages 19–41. Addison Wesley, 2003. 225

S. Zlatanova. On 3d topological relationships. In *Proceedings of the 11th International Workshop on Database and Expert Systems Applications*, page 913, London/Greenwich, UK, 2000. 65