Vrije Universiteit Brussel

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

Conceptual Modeling of Complex Objects for Virtual Environments *A Formal Approach*

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

Wesley Bille

Academiejaar: 2006 - 2007

Promotor: Prof. Dr. Olga De Troyer

Samenvatting

Virtuele Realiteit (VR) wordt gebruikt in verschillende domeinen en voor verschillende doeleinden. De technologie heeft de laatste jaren enorm aan populariteit gewonnen. Door de groeiende interesse in VR werden een groot aantal tools voor het bouwen van VR applicaties ontwikkeld. Ondanks de aanwezigheid van deze tools blijkt een grote achtergrondkennis omtrent VR technologie nog steeds noodzakelijk om de gewenste virtuele omgeving te bouwen. Momenteel laat geen enkele tool de ontwerper toe om de virtuele omgeving te specificeren in termen van domeinconcepten. Bij de creatie van een VR applicatie moeten bijvoorbeeld de objecten van het probleemdomein eerst vertaald worden in VR primitieven. Dit is één van de redenen waarom het ontwikkelen van een VR applicatie zo complex, tijdrovend en duur is.

Om deze problemen op te vangen heeft de onderzoeksgroep WISE van de Vrije Universiteit Brussel de VR-WISE benadering ontwikkeld. Deze nieuwe benadering voor het ontwikkelen van virtuele omgevingen introduceert een expliciete conceptuele modelleerfase in het ontwikkelingsproces van een VR applicatie. Conceptueel modelleren biedt een mechanisme om te abstraheren van het implementatieniveau. Dit reduceert de complexiteit bij het ontwikkelen van een VR applicatie. Bovendien kan zo'n abstractielaag het specifieke VR vakjargon verbergen zodat op deze manier geen specifieke VR kennis nodig is om het conceptueel model te creëren. Hierdoor kunnen mensen zonder technische kennis (zoals de klant of de eindgebruiker) betrokken worden in de ontwikkeling. Dit komt de communicatie tussen de ontwikkelaars en de andere deelnemers binnen het project ten goede. Doordat de klant dichter bij de ontwikkeling van de VR applicatie betrokken wordt is het mogelijk om misverstanden en onjuistheden vroeger te ontdekken.

In deze thesis wordt dieper ingegaan op het conceptueel modelleren van complexe objecten. Complexe objecten zijn een belangrijk onderdeel van een VR applicatie. Objecten in de reële wereld bestaan vaak uit verschillende onderdelen. De manier waarop deze onderdelen aan elkaar vasthangen bepaalt eveneens hoe deze objecten kunnen bewegen ten opzichte van elkaar. Daarom wordt in deze thesis een verzameling van hoog-niveau modelleerconcepten voor het modelleren van complexe objecten in de context van de VR-WISE benadering voorgesteld. Deze modelleerconcepten bieden een abstractieniveau bovenop bestaande VR primitieven voor connecties tussen onderdelen. Voor alle modelleerconcepten die geïntroduceerd worden in deze thesis werd een grafische notatie ontwikkeld alsook een formele definitie van zowel syntax als semantiek. De grafische notatie helpt in het vormen van een mentaal model van het conceptueel model en vergemakkelijkt ook de specificatie van het conceptueel model. De formele definities leggen de semantiek van de verschillende modelleerconcepten ondubbelzinnig vast en laten toe om ondubbelzinnige conceptuele specificaties te bouwen. Bovendien biedt de formalizatie het voordeel dat ze aangewend kan worden om te redeneren over de virtuele omgeving. Eveneens kunnen tools ontwikkeld worden die de stappen beginnend bij de conceptuele specificatie tot de corresponderende virtuele omgeving, ondersteunen. Een dergelijke prototype implementatie werd ontwikkeld om de haalbaarheid van de benadering aan te tonen.

Abstract

Virtual Reality (VR) is used in lots of different domains for different purposes. The technology has gained a lot of popularity during the last decennia. Together with the growing interest in VR, a lot of different software tools were developed which allow building VR applications. However, most tools available nowadays require considerable background knowledge about VR technology in order to create the desired virtual environment. It is impossible for a developer to specify the virtual environment immediately in terms of domain concepts, i.e. when creating a VR application the objects from the problem domain have to be translated into VR building blocks. This is one of the reasons why developing a VR application is complex, time-consuming and expensive.

To cope with these problems, the research group WISE at the Vrije Universiteit Brussel has developed the VR-WISE approach. This new approach for developing virtual environments introduces an explicit conceptual design phase in the development of a VR-application. As conceptual modeling introduces a mechanism to abstract from implementation details, it will reduce the complexity of developing a VR application. In addition, such an abstraction layer can also hide the specific jargon used in VR and then no special VR knowledge will be needed for making the conceptual design. Therefore, also non-technical people (like the customer or the end-user) can be involved in the development and this will improve the communication between the developers and the other stakeholders. In addition, by involving the customer more closely in the design process of the VR application, earlier detection of design flaws is possible. All this could help in realizing VR applications in a shorter time.

In this dissertation, we have focused on the conceptual modeling of complex objects. Complex objects are an important issue in VR applications. Like in the real world, objects are often composed of different parts, and the way the parts are connected influences the way the complex objects can behave. Therefore, in this dissertation we propose a set of high-level conceptual modeling concepts for specifying complex objects in the context of the VR-WISE approach. The modeling concepts proposed provide an abstraction layer on top of existing VR-primitives for connections. For all modeling concepts introduced, a graphical notation has been developed and a formal specification is defined for the syntax as well as for the semantics. The graphical notation helps in quickly building a mental model of a specified conceptual model and eases the specification of the conceptual model itself. The formal specification unambiguously defines the semantics of the different modeling concepts and therefore allows building unambiguous conceptual specifications. Furthermore, it has the advantage that it provides the basis for doing intelligent reasoning over the virtual environment. In addition, tools can be developed to assist in the translation from the conceptual specifications towards a working virtual environment. A prototype has been implemented to show the feasibility of this translation.

Acknowledgement

First of all, I would like to thank my promotor *Prof. Dr. Olga De Troyer*. I would like to thank her for her believe in me and the support she gave me when producing this dissertation. In spite of her time-consuming task of managing the Web and Information Systems Engineering lab she still reserved some of her rare time to proofread my chapters and to give me very useful comments and advises. She also gave me the right motivation in order to complete this PhD.

Secondly I would like to thank the members of my jury, *Prof. Dr. Stefano Spaccapietra*, *Dr. Clive Fencott*, *Prof. Dr. Robert Meersman*, *Prof. Dr. Dirk Vermeir* and *Dr. Frederic Kleinermann*, for being part of my jury. They all provided me very useful comments and suggestions in order to improve my work.

Next I also want to thank my collegues and ex-collegues at WISE: *Sven Casteleyn*, *Peter Plessers*, *Raul Romero* and *Abed Mushtaha*. Special thanks goes out to my collegues *Bram Pellens* and *Dr. Frederic Kleinermann* for the excellent cooperation and the interesting discussions and nice times we had.

Furthermore, I also would like to thank my family and friends for their support during this hard time.

The last word of thanks is reserved to two special people in my life, my girlfriend and my daughter. I want to thank my girlfriend, *Linsday Decoopman*, for enduring me during my moments of bad mood and for giving me the necessary motivation. Although she's might not yet be conscious of it, thanks goes to my daughter *Febe*, who was born on the 14th of February 2007, for giving me the strength to continu. Thank you Lindsay and Febe!

Contents

List of Figures

List of Tables

Part I

Introduction, Background and Informal Description

Chapter 1

Introduction

Virtual Reality (VR) is a technology allowing users to interact with a computer-simulated environment. Nowadays, VR is used in lots of different domains and for different purposes. VR can be used to simulate an environment very close to the real world, as it is the case for flight simulators, or building prototypes in industry. But VR can also be used to simulate artificial world environments. This kind of environments is often found in computer games. In any case, the use of VR has gained a lot of popularity during the last decennia.

A number of software tools are available today to assist in the development of VR applications. However, no matter how powerful these applications are, they require a considerable VR background knowledge. This way, the development of VR applications is still a specialized and tedious task, which hinders the application of VR in areas and domains that could also benefit from its use (like e.g., e-commerce and e-learning). Also, the design phase in the development of a VR application is usually a very informal activity. This often results in mismatches between the expectations of the customer and the actual VR application delivered. For these reasons, this dissertation is concerned with the problem of modeling virtual environments on a high level of abstraction. More in particular, we are focusing on modeling complex objects for virtual environments. Note that the term modeling used in this dissertation refers to the activity performed during the design phase of software development. We will also use the term specifying (respectively specification) as a synonym for modeling (respectively model), as the term modeling is also used in the VR community to denote the process of shaping objects.

1.1 Research Context

Virtual Environments can be seen as computer-generated environments that create the effect of an interactive three-dimensional world in which objects have a sense of spatial and physical presence and can be manipulated by the user as such. The user can pick up objects, turn or move them, etc. Virtual environments can range from low-end virtual environments like web-based X3D models [1] to high-end virtual environments like fully immersive environments such as a CAVE [16]. The type of VR we are targeting in this dissertation can be categorized as desktop-based VR. Desktop-based VR are virtual environments displayed on a normal computer screen and manipulated by means of classic input devices like a keyboard or mouse. Note that nowadays also some more sophisticated input devices are used for desktop VR. Some examples are a 3D mouse or a glove.

A number of techniques and tools are available today that can be used for developing VR applications. These tools can be categorized in two categories. The first category consists of so called toolkits (like ODE [55], Havok[1], Ogre3D[2]). These toolkits are programming libraries that provide a set of functions that can be used by a skilled programmer to create VR applications. The second category are the authoring tools (like 3D Studio Max [41]). These are complete software environments with graphical user interfaces for building virtual environments without the need for detailed programming. Current practice for developing a VR application is that first an authoring tool is used to create the 3D content for the virtual environment and afterwards this content is imported in a toolkit where the code for behavior is added. Note that some of the recent authoring tools also provide a simple scripting language which makes it possible to develop simple behaviors for the 3D content.
However, although the development of a VR application is supported by a number of powerful tools, it remains a specialized and tedious task to develop a VR application. A number of problems arise when using the currently available tools:

- When using the available tools, a considerable background knowledge about VR technology is required.
- None of the available development tools allow the developer to specify the virtual environment in terms of domain concepts.

[1]http://www.havok.com, accessed 6th of June 2007
[2]http://www.ogre3d.org, accessed 6th of June 2007

We will discuss these problems in more detail in section 1.2.

In order to give an answer to these problems the research group WISE at the Vrije Universiteit Brussel has developed a new approach for developing virtual environments, called the VR-WISE approach. This approach introduces an explicit conceptual design phase in the development process of a VR-application. It provides a set of high-level modeling concepts to allow modeling a VR application using knowledge from the application domain. When developing this set of high-level modeling concepts to support the design process a number of requirements have been taken into account:

1. **Intuitiveness:** The set of high-level modeling concepts needs to be intuitive to a non-VR expert.

2. **Expressive power:** The modeling concepts provided by the approach must be expressive enough to allow deriving an implementation from the conceptual design. So the expressive power of the modeling concepts needs to be high enough to be able to serve as input for the implementation process, which we call the code generation.

3. **Unambiguousness:** First of all, the modeling concepts must be unambiguous from the perspective of the designer. For the designer it is important that he understands the semantics of a conceptual model. Next, unambiguousness is also needed from the perspective of the code generation; otherwise it would not be possible to automatically generate code.

To ease the use of the modeling concepts and to enhance the communication between designers, programmers and other stakeholders in a project, a graphical notation for the modeling concepts has also been developed.

The work performed for this dissertation was done in the context of VR-WISE. The focus of the research is on complex objects. Complex objects are objects composed of two or more components (like most real-world objects). Most of the objects in a virtual environment are also complex. Therefore, a set of high-level modeling concepts that can be used for modeling complex objects inside the context of the VR-WISE approach was needed.

1.2 Problem Statement

Although the creation of VR applications is supported by a number of software tools (see section 1.1), the development stays a specialized and tedious task. These tools are very powerful but they cause a number of problems. This dissertation does not claim to present a complete solution to all problems related to the development of virtual environments. Since most real world objects are composed of several components and this often needs to be reflected in the VR application, the modeling of complex objects is an important issue when developing a VR application. Therefore, in this dissertation we focus on modeling complex objects in the context of the VR-WISE approach. Hence, we will propose an answer to the following problems from the viewpoint of complex objects.

- **Problem 1:**
  The available tools for developing VR applications require a considerable knowledge about VR technology. Current practice when developing a VR application is that first the static part is created by means of an authoring tool. Next, the outcome of the static part modeling is imported in a toolkit where the code for behavior is added. Toolkits require detailed programming and thus require a considerable programming background while authoring tools make use of a typical VR vocabulary which is only familiar to VR specialists.
  In addition, the translation of the domain objects needed in the virtual environment (for example a car) into a combination of VR primitives (such as boxes, cylinders, . . .) and constraints (like the type of connection needed between several primitives) is not an easy task. Take for example two objects in the domain that are connected over a center of motion. In some toolkits this type of connection must be translated as a spherical joint type. None of the current VR development tools give the developer support for this task or allow him to specify the content of a virtual environment in terms of domain concepts. To illustrate this, take VRML [29] or X3D [1]. Although VRML and X3D allow the creation of 3D content without having to deal with low-level details of the platform or rendering process, the developer still has to specify the concepts using low-level primitives.

- **Problem 2:**
  The design phase in the development process of a VR application is usually a very informal activity. Developers usually start with the so-called back-of-the-envelope approach. First they do some brainstorm sessions, then they sketch some general shapes to get a clearer idea of what is being created, then they create some early prototypes in a CAD-environment. Few formal techniques in the context of VR

exist to support the design phase effectively (like UML [21] in the context of classical software engineering). A fully developed systematic approach that uses the output of the design phase as the input for the implementation phase does not exist.

- **Problem 3:**
  There does not exist a formal basis for discussing the design of a Virtual Environment between the different stakeholders of a project. As we already discussed in problem 2, based on some brainstorm sessions, notes and sketches are made. However, natural language and sketches are informal, ambiguous and often incomplete. The consequence is that this easily leads to misunderstandings. Because of these misunderstandings, part of the VR application might need to be remodeled or even rebuild.

1.3 Goal

In this dissertation we aim to develop an approach for modeling complex objects and shapes for virtual environments that:

- requires little VR background knowledge from the user.
- can serve as a communication basis between designers, programmers and other stakeholders of the project.
- has a formal foundation that unambiguously defines the modeling concepts and allows building unambiguous conceptual specifications.
- provides the possibility to do some intelligent reasoning over the virtual environment.

1.4 Approach

The approach taken in this dissertation to tackle the problems described above is to introduce a set of high-level modeling concepts for specifying complex objects for virtual environments. The proposed set of modeling concepts fits in the general VR-WISE approach.
The use of high-level modeling concepts that allow specifying complex objects and shapes in term of domain concepts instead of low level VR primitives must ensure that also domain experts can be involved in the development. To reach the other goals formulated in section 1.3, we also took into

account their expressive power, intuitiveness and unambiguousness when we developed this set of modeling concepts.
The VR-WISE approach distinguishes two levels. On one hand we have the *conceptual domain specification* describing the concepts of the application domain. On the other hand we have the *conceptual world specification* containing the actual conceptual description of the virtual environment that needs to be build. The conceptual world specification actually contains instances of the concepts described in the conceptual domain specification.
Since we follow the VR-WISE approach, the approach for modeling complex objects for virtual environments introduced in this dissertation also follows this two-level approach. In the conceptual domain specification, we describe which concepts from the application domain are complex and how they are connected to each other and how they are constrained. Next, at the conceptual world specification, we specify the instances of the complex concepts described in the conceptual domain specification.

The set of modeling concepts for modeling complex objects has been divided in several categories. The first category contains the so-called *connection relations*. This category contains relations that can be used to physically connect two objects to each other and also limit the degrees of freedom for the movement of the connected objects with respect to each other. We have identified three connections relations: the *connection point relation*, the *connection axis relation* and the *connection surface relation*. These relations respectively specify a connection over a 'center of motion', an 'axis of motion' and a 'surface of motion'.
The second category of modeling concepts are the so-called *contraints*. These constraints allow further restriction of the position and orientation of connected objects with respect to each other. All the constraints are modeled on top of a connection relation between two objects. We introduce the *hinge constraint*, the *joystick constraint* and the *slider constraint*. The names of the constraints are metaphor-based which should enhance the intuitiveness of the modeling concepts for non-technical persons.

In the case of complex objects, all the components connected together keep their own identity and can all be manipulated individually in the virtual environment as far as their connections and constraints allow. However, we can think of cases where the different components may be melted together and thus loose their identity. The result of melting objects together to form one whole is called a *complex shape*. Actually, when modeling complex objects we are concerned with the physical structure of the complex object.

When modeling complex shapes we are concerned with the representation of the complex shape in the virtual environment. For this category of modeling concepts we have based our approach on the relations provided in the domain of *Constructive Solid Geometry* (CSG) [52]. CSG is a technique used for solid modeling (a solid model is just another name for what we call an object). This technique uses a set of boolean operators to create complex shapes. The CSG technique allows creating very complicated geometries with a number of very simple operators. Therefore in our approach we also introduced the *intersection relation*, the *union relation* and the *difference relation*.

Next to complex objects and complex shapes it is also possible that a number of objects are constrained with respect to each other without being physically connected. We call such a group of objects a *connectionless group of objects*. Some examples in the real world are a magnetic field between two objects, or a coffee cup that can only be positioned on a saucer. Therefore we also introduced a number of constraints which can be used for this purpose.

Following the general VR-WISE approach we also developed a graphical notation for the modeling concepts we introduce in this dissertation. The graphical notation follows the graphical notation of the general VR-WISE approach being an icon-based graphical representation. The relations are graphically represented in the same way but with a unique icon to identify the type of the relation and to make it easier to recognize the different relations inside a conceptual model.

When modeling concepts are not formally defined, they can be interpreted in different ways leading to ambiguity. Therefore a formal foundation has been defined for the work described in this dissertation. Such formalization introduces a number of new opportunities and advantages. One of the new opportunities is that it allows reasoning over the virtual environment. Our formalization has been defined using F-Logic which is a full-fledged logic following the object-oriented paradigm.

1.5 Contributions

With the work presented in this dissertation we have made a number of contributions. The main contributions follow from the research which has been done specifically in the context of this dissertation [3].

- We present a set of high-level modeling concepts that can be used to specify complex objects for a virtual environment. These modeling concepts can be divided into several categories. The first category is the one of connection relations that allow to specify how two objects are connected to each other and how they are allowed to behave with respect to each other. The second category contains modeling concepts that allow further constraining the behavior of two connected objects. Next we also introduce a number of constraints that can be used to constrain the position and orientation of unconnected groups of objects relative to each other.

- Our research also presents a number of high-level modeling concepts to specify complex shapes. Complex shapes are formed by combining a number of simple and/or complex shapes together to form one object (in contrast to complex objects where each component can still be manipulated individually).

- For all the presented modeling concepts in this dissertation we also define a graphical notation which is consistent with the graphical notation used in the rest of our VR-WISE approach.

- To unambiguously specify the modeling concepts we also present a formalization. This allows building conceptual specifications of virtual environments that are unambiguous. Therefore, first the VR-WISE approach, which is used as a framework for the work in this dissertation, has been formalized. Next the modeling concepts supporting the specification of complex objects and complex shapes have been formally defined.

- The formal foundation allows us to do some intelligent reasoning. The conceptual level can serve as a semantic annotation of the virtual environment while the formalization mechanism can be used for querying it.

- Finally we have developed a prototype tool. This tool serves as a proof-of-concept for the ideas presented in this dissertation. We developed an extension to Microsoft Visio that can be used to draw the

[3]The research in the context of this dissertation was mainly funded by the Fund for Scientific Research - Flanders (FWO).

graphical representation of the conceptual specifications. Next we also implemented a tool that takes care of code generation for the complex objects. This has been integrated into the overall tool taking care of the complete process of translating the conceptual specification into a working virtual environment.

1.6 Outline

This dissertation is structured in three parts.

Part I: Introduction, Background and Informal Description

Chapter 2 contains background and related work. It gives a general introduction to Virtual Reality and also to conceptual modeling. A number of existing conceptual modeling languages are discussed. Advantages and disadvantages of their use for modeling virtual environments are discussed. For the related work an overview of state-of-the-art modeling approaches for virtual environments is given. The described approaches can be divided in two main categories, namely academic approaches and commercial approaches.

Chapter 3 gives an informal description of the VR-WISE conceptual modeling approach. The VR-WISE approach introduces an explicit conceptual design phase inside the development process of a VR-application. The research proposed in this dissertation is part of this approach. First, a general overview of the approach is given. Next, an overview of existing modeling concepts that can be used inside the approach for modeling simple objects on a higher level of abstraction are discussed. Some attention is also paid to high-level modeling concepts for modeling behavior inside the VR-WISE approach. However, behavior modeling is the topic of another PhD-dissertation.

Chapter 4 introduces a set of high-level modeling concepts that can be used to specify complex objects and complex shapes at the conceptual level in the context of the VR-WISE approach. Most real-world objects are assembled of several components. Components connected to each other form a complex object. Usually, all of these components keep their own identity and hence it should be possible to manipulate them individually in the virtual environment as long as their connections and constraints allow this. In this chapter attention is paid to several categories of modeling concepts for complex objects: connection relations, constraints on connection relations and constraints for connectionless groups of objects. We also introduce a set of high-level modeling concepts that can be used to specify complex shapes. The components of complex shapes, in contrast to the components of complex objects, are melted together to form one object.

Part II: Formal Definitions

Chapter 5 gives an introduction to F-logic. F-logic is a full-fledged logic that will be used in this dissertation to give a formal specification of the modeling concepts developed for modeling simple as well as complex objects. First F-Logic is described by means of some examples. Next, the formal syntax is introduced. Finally, we give a clarification why F-Logic has been chosen for the formalization.

Chapter 6 gives the formalization for a number of fundamentals which are needed in the rest of our formalization. This chapter formalizes concepts as *point*, *rotation*, ...

Chapter 7 formalizes the modeling concepts for modeling simple objects. The basic modeling concepts from the VR-WISE approach such as *concept*, *instance*, *spatial relations* and *orientation relations* are formally defined.

Chapter 8 gives the formal definition for the connection relations which were introduced in chapter 4. The *connection point relation*, *connection axis relation* and *connection surface relation* are formalized.

Chapter 9 gives a formalization for the following constraints: the *slider constraint*, *hinge constraint*, *joystick constraint*, *fixed relative position constraint*, *fixed relative orientation constraint* and *positioning constraint* are formalized.

Chapter 10 gives the formal definition of the modeling concepts that can be used to model complex shapes.

Chapter 11 finally gives some examples of possible applications of the formalization. Reasoning over the conceptual model and consistency checking are illustrated.

Part III: Implementation, Use Case and Conclusions

Chapter 12 describes the prototype tool that has been developed. This tool serves as a proof of concept for the ideas and results presented in this dissertation.

Chapter 13 give a use case of the material introduced in this dissertation. The use case illustrate the possibilities of the modeling concepts introduced for modeling complex objects and shapes for virtual environments. This is done by means of an elaborated example, namely the modeling of a mechanical welding robot.

Chapter 14 provides a summary of the results presented in this dissertation. Limitations of the presented approach are discussed and an overview of the achievements is given. Also possible future work and extensions to the presented approach are discussed.

Chapter 2

Background and Related Work

In the previous chapter we have introduced this dissertation. We have given the research context in which this dissertation is situated. We have presented a number of problems and identified the goals of this dissertation. Next we have briefly introduced the approach we have chosen in order to reach these goals. Finally we have outlined the advantages of this approach and we have listed the contributions of this dissertation to the research field. In this chapter, we present the necessary background knowledge and related work which are both required to place this dissertation in a broader perspective. In section 2.1 we give a brief introduction to Virtual Reality. Next in section 2.2 we discuss the domain of conceptual modeling. A short overview of existing conceptual modeling languages in the domains of software engineering and information systems is given. Next a discussion follows about the usability of these languages for modeling VR. As related work to this dissertation, state-of-the-art modeling approaches for VR are discussed. In section 2.3 we outline a number of academic modeling approaches. In section 2.5 we discuss a number of commercial modeling approaches. All these approaches are discussed in the context of modeling complex objects. Finally section 2.6 concludes this chapter with a short summary.

2.1 Virtual Reality

The term Virtual Reality (VR) covers a broad range of applications and technologies going from low-end VR like web-based X3D models [1] to high-end VR like fully immersive environments such as a CAVE [16]. Virtual

Reality, Virtual Environments, Virtual Worlds, . . . these are all terms used to denote more or less the same thing in many different contexts. Most of these terms correspond to each other but there may be some small differences. In this dissertation we will use the term virtual reality environments or simply virtual environments.

Essentially, VR is about the navigation and manipulation of 3D computer-generated environments [65]. John Vince states the following general accepted statement about VR: *VR is about creating acceptable substitutes for real objects or environments and is not really about constructing imaginary worlds that are indistinguishable from the real world.*
During the last decennia people came with different definitions for VR. However, all of these definitions must be seen in the context of the research in which they are stated. Some definitions talk about the sensorial modalities such as tactile, smell and taste [12]. This more sophisticated kind of VR is not the one we are targeting in this thesis. The description of VR that fits most the type of VR we are targeting in our research is the one coming from Aukstakalnis and Blatner [4]:

Virtual reality can be used to fly through three-dimensional environments that represent extremely complex data, reach out and manipulate these representations with your hand, or experience visiting a world that would be physically impossible in our own reality.

We see Virtual Environments as computer-generated environments that create the effect of an interactive three-dimensional world in which objects have a sense of spatial and physical presence and can be manipulated by the user as such. The user can pick up objects, turn or move them, etc.

2.1.1 A short history of Virtual Reality

The origin of the term Virtual Reality is unsure. In several books, Morton Heilig is called the father of Virtual Reality. In the 1950s Heilig believed that by expanding a cinema to involve not only sight and sound, but also taste, touch and smell, the viewer could be drawn into the onscreen activity. He called this idea the *experience theatre*. In 1962 he build a prototype machine called Sensorama together with a number of films that could be displayed while stimulating multiple senses of the viewer. The Sensorama simulator included motion, color, stereo sound, wind effects and a vibrating seat.
In 1968, Ivan Sutherland created with the help of his student Bob Sproull

what is widely considered as the first virtual reality head-mounted display (HMD) [58]. This device was very primitive in terms of user interface as well as in terms of realism. The graphics were just wireframes. The HMD was attached to a mechanical arm suspending from the ceiling in order to be able to track the head movements and to cope with the weight of the HMD. The name of the device was inspired by its appearance, namely the *Sword of Damocles*.
Mid-1970s Myron Krueger came with VideoPlace. With VideoPlace the computer responded to the gestures of the users by interpreting their actions. The movements of the users were recorded by means of video cameras and transferred to the silhouette representations of the users in the *Artificial Reality* environment.
However, Jaron Lanier claims that he coined the term *Virtual Reality* in the early 1980s. It was also at that time that he founded VPL Research which was the first company to sell VR products. In 1987 VPL came with DataGlove [69] which is know as the first commercial version of gloves used to interact with computers.
In 1992 Sense8 Co. developed WorldToolKit which is a library consisting of C-functions written for VR applications and still existing today. Also in 1992, researchers from the University of Illinois at Chicago presented the first CAVE, a VR theatre with walls onto which images are projected surrounding the users with sights and sounds. Users can walk freely in the room and interact with and manipulate 3D virtual objects.
In the early 1990s a lot of VR companies were founded but by mid-1990s VR reached a critical point. However, late 1990s it came to a rebirth of Virtual Reality thanks to tremendous improvements in PC hardware. Nowadays, there are a lot of commercial VR hardware devices and VR software going from development tools to end-user applications.

2.1.2 Different types of VR-applications

Two main characteristics of VR applications are the interactivity and the immersion experienced by the user. Note that Burdea [12] also takes imagination as a characteristic of a VR application. However, we will not take this into account. Immersion and interactivity can reach different levels according to the variety in technologies and hardware used. Using a regular screen gives another level of immersion than when using a cave. Based on these characteristics different types of VR applications can be distinguished. We will only list the most known and most important types here.

- **Desktop VR:** This is the use of animated interactive 3D graphics to build Virtual Worlds with desktop displays [50]. In this type of VR the user explores an interactive, real-time virtual environment on a conventional display. Examples of desktop VR are computer-games and VRML or X3D worlds on the Web

- **Immersive VR:** CAVE environments or head-mounted displays are often used for immersive VR systems. Using these technologies, the level of immersion is much higher than for example with desktop VR.
- **Augmented Reality (AR):** In [5] Azuma defines Augmented Reality (AR) as follows. While immersed in Virtual Environments, the user cannot see the real world around him. When using AR, the user can see virtual objects placed inside the real world. AR can be seen as a supplement of reality rather than replacing it, which is the case with the other types of Virtual Environments. According to Azuma, AR has three characteristics. It combines objects, real as well as virtual, in a real environment. It is interactive in real-time. And finally it aligns the real and virtual objects with each other.

As already stated before, in this dissertation we are targeting desktop VR.

2.2 Conceptual Modeling

Conceptual modeling can be defined as the activity of building a model of the Universe of Discourse (UoD) in terms of concepts that are familiar to domain experts and free from any implementation details. The Universe of Discourse is the part of the world under consideration. In other words, conceptual modeling can be seen as modeling a human's conceptualization of some UoD.

Conceptual models are mostly built early in the development stages of a system before the system design and implementation phases. Conceptual models can be used for several purposes and advantages:

- Improve the understanding of the concepts in the domain of discourse and produce information models that are clear to designers, programmers, customers, etc.
- Extracting parts of the domain under consideration wich are of importance to the project.
- Provide a communication platform between the various stakeholders (such as users, developers, customers, programmers, etc.) in the project.
- Detecting misunderstandings, errors and missing information early in the system development.

Conceptual modeling is gaining importance since software systems become more complex and the problem domains are sometimes far away from the background knowledge of the developers. The term conceptual modeling originally emerged from the domain of databases. In the field of database systems, conceptual database design is the process of constructing a model of the information used in an enterprise, independent from all physical considerations [15]. This means that a conceptual data model is created containing the part of the enterprise information that is of interest for the system to be build. The data model is built using the information from the requirements analysis and throughout the process of developing the conceptual data model, the model is continually validated against the information from the requirements analysis. Following the domain of databases, conceptual models also became popular in the domains of software engineering and knowledge engineering.

Conceptual modeling requires notations, tools and techniques for representing information and processes about a domain of interest. In the following subsections we will look to some popular conceptual modeling languages developed in the domains of software engineering and information systems. After introducing these languages we will discuss their usability for building conceptual models for virtual reality applications.

2.2.1 UML

The Unified Modeling Language (UML) [21] can be seen as a conceptual modeling approach that provides a set of notations that facilitates the development of a software project and makes it easier for the various stakeholders in a project to communicate about the project. It actually organizes the design process in a way that analysts, clients, programmers and others involved in system development can understand and agree on. UML uses a combination of different diagrams in order to model all the different aspects of a software application. The main diagram notations of UML are:

- **Class diagrams:** A class diagram describes the type of objects in the system and the various kinds of static relationships that exist among them. Actually, a class diagram is a graphic view of the static structure of a system. Note that occasionally object diagrams are used to show individual objects and their relationships. These object diagrams are sometimes used for documenting test cases.

- **Interaction diagrams:** Interaction diagrams are models describing the behavior of a system in terms of the interactions between objects. There are two kinds of interaction diagrams: sequence diagrams and communication diagrams [1]. Both diagrams can be used to show the interaction flow through the use of messages between objects. Sequence diagrams focus on the order of the messages as they show the interactions in a sequential order. Collaboration diagrams more focus on showing the collaboration between objects rather than on showing the time sequence of the interactions.

- **Behavior diagrams:** Behavior diagrams describe the behavior of a system in terms of its state changing during execution. State machine diagrams [2] are an example of behavior diagrams. A state machine diagram describes all the possible states an object may be in and how the object's state changes by means of translations.

[1]For UML versions earlier than UML 2.0 communication diagrams were called collaboration diagrams.

[2]State machine diagrams were previously called state diagrams.

2.2.2 ER

Entity-Relationship (ER) [13] modeling has been designed to facilitate database design by developing a high-level, conceptual data model. The main purpose of a conceptual data model is to support the user's perception of the data model and to conceal the more technical aspects associated with database design. The basic concepts of the Entity-Relationship model include *entity types*, *relationship types* and *attributes*. This way, an ER model is expressed in terms of entities having attributes and participating in relationships. The fact that a *Car* is manufactured by a *Manufacturer* is modeled as shown in the ER model in figure 2.1.

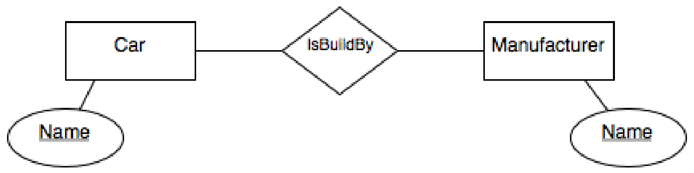


Figure 2.1: Example of an ER model

Car and *Manufacturer* are entity types. Entity types are concepts which are identified in the problem domain as having an independent existence. *IsBuildBy* is called a relationship type. A relationship type is a meaningful association among entity types. Next, *Car* and *Manufacturer* have one attribute each, which is their *Name*. At the same time, these attributes are the primary keys for their entity types. This is indicated by underlining the name of the attribute. This means that the *Name* uniquely identifies individual occurences of the entity type.

2.2.3 ORM

Object-Role Modeling (ORM) [27, 28] is a semantic modeling approach which views the world as objects playing roles. ORM uses natural language and intuitive diagrams for data modeling. Unlike ER, ORM includes a step-by-step design procedure. This procedure starts with specific examples of information related to the application domain. These information examples are expressed in terms of elementary facts. Elementary facts are simple sentences noting what kind of objects are present, how they are identified and what roles they play. Afterwards, these facts are diagrammed.

An example of such an elementary fact is the following sentence:

The Car golf is build by the Manufacturer Volkswagen.

The diagram derived from the above fact is shown in figure 2.2. The arrow on the first part of the binary predicate between Car and Manufacturer indicates that a Car can have at most one Manufacturer. The dot on the Car entity type indicates that each Car has a Manufacturer.

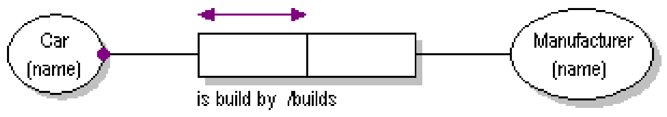


Figure 2.2: Example of an ORM diagram

2.2.4 Discussion

As stated in [22], a VR application can often be decomposed into four modules. The first module is called *form*. This module describes the appearance of virtual objects, their structure, their relation with other objects and other physical properties like mass. The second module is called *function*. Function refers to primitive actions required for virtual objects to carry out high-level behaviors. The third module is the *behavior* module. This module describes how a virtual object dynamically changes over some period of time. The last module is the *interaction* module defining the interaction between virtual objects as well as between the user and the virtual objects. So the form module describes the static part of a VR application while the behavior, function and interaction modules describe the dynamic part.

ORM and ER more or less have the capability to model the form module of a VR application. However, ORM is lacking modeling concepts in terms of expressiveness towards VR modeling. We illustrate this by means of an example. Figure 2.3 illustrates the modeling of a connection between a base concept and a handle concept.

When we look to the ORM model in figure 2.3, we can see that the fact that a handle is connected to a base is expressed by a binary relation between a *Handle* entity type and a *Base* entity type. These entity types represent

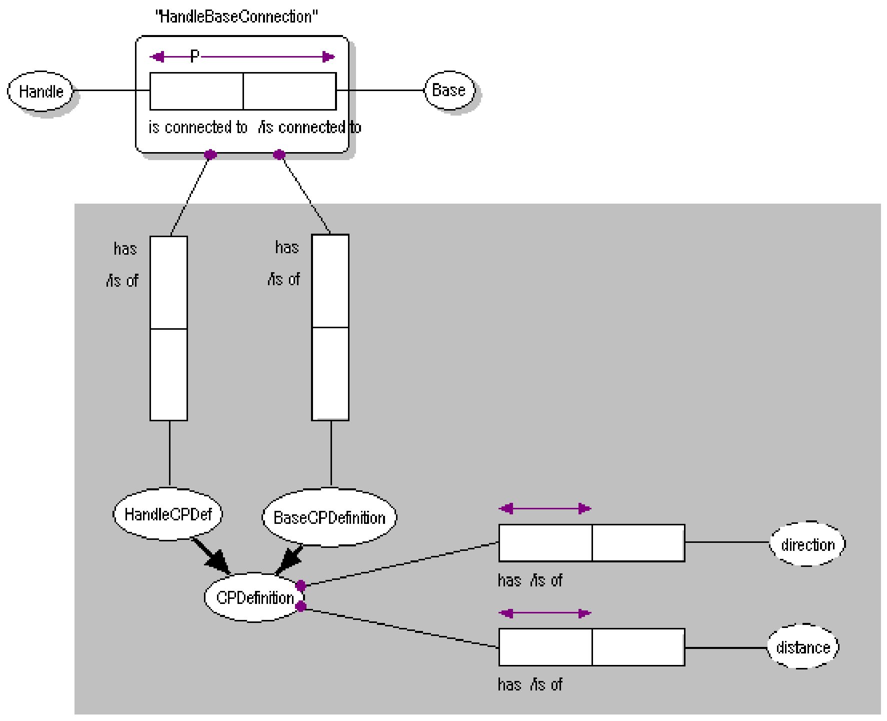


Figure 2.3: Example of the modeling of a connection between two objects using ORM.

the domain concepts for the virtual environment that needs to be modeled. To express the exact semantics of the relation between the handle and the base, the grey part of figure 2.3 is used. We can see that the connection relation has one connection point definition for the handle (called HandleCPDef) and one connection point definition for the base (called BaseCPDef). Each of these connection point definitions has a direction and a distance. Note that we will not explain here the meaning of such a connection point definition as this returns later in this dissertation.
This example illustrates a number of problems. First of all, everything is represented in the same way. Domain concepts such as Handle and Base are represented in the same way (using entity types) as for example the more abstract concept connection point definition. In this way, it is difficult to recognize in the model the domain concepts. Next, the semantics of

the connection relation between the Base and Handle entity type is actually given in the grey part of the figure. However, when connecting two other domain concepts in the same way, this semantics needs to be expressed again. In order to solve this neatly we would need to extend ORM on the meta-level with a new type of relation and the necessary semantics. Therefore, plain ORM is not appropriate for modelling these issues. The case for ER is similar.
Furthermore, since the VR-WISE approach is targeting the modeling of form as well as behavior, ORM and ER are not suited without extensions. Note that various extensions have been proposed for ORM for object-oriented and dynamic modeling [7, 60]. Nevertheless, the focus of ORM is on data modeling.

From this viewpoint, UML is more expressive than standard ORM and ER since it offers a number of diagrams to model the system dynamics. Again, although UML can be used to model a VR application from a software engineering point of view, it is also lacking in terms of expressiveness and intuitiveness towards VR modeling for domain experts. Figure 2.4 shows the same example of modeling a connection between two concepts.

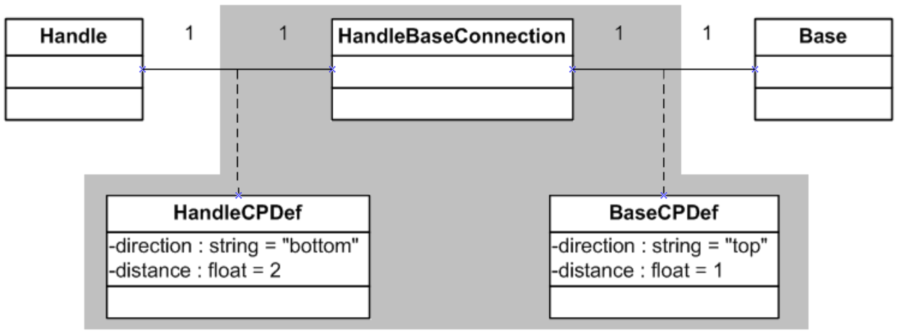


Figure 2.4: Example of the modeling of a connection between two objects using UML.

Again, from the UML model represented in figure 2.4 a number of problems can be identified. As was the case for ORM, all types of information are represented in the same way. Concepts are represented as classes and so is the connection. Next, in the grey part of the figure, the semantics of the connection relation is defined. Again, if we need to model a similar connection between two elements, similar semantics need to be defined.

Table 2.1: Usability of conceptual modeling languages for VR modeling

	Form	Function	Behavior	Interaction
UML	+/-	+/-	+/-	+/-
ER	+/-	-	-	-
ORM	+/-	-	-	-

In contrast to ORM and ER, UML contains the concept of *stereotypes*. A stereotype is a new type of modeling element that extends the semantics of the metamodel. Stereotypes must be based on certain existing types or classes in the metamodel. So stereotypes allow you to extend the vocabulary of UML so that you can create new model elements derived from existing ones and which are suitable for the problem domain. Extensions made by means of stereotypes appear as basic building blocks in UML. So actually they can be seen as first class citizens to UML. In principle, we could use these stereotypes to define new modeling elements needed for VR. However, these stereotypes also have some disadvantages. As stated in [8], stereotypes increase the complexity of the base language (in this case UML). In addition it is not clear if the concept of stereotypes will be powerful enough to define all the modeling concepts needed. On the other hand, UML also contains much more core concepts than needed for our purpose.

As mentioned in [44], table 2.1 summarizes the possibilities of UML, ER and ORM for modeling VR from the perspective of a non-VR specialist. '+/-' means that the modeling language can partially model the module (although it may lack expressiveness and/or intuitiveness) while '-' indicates that the conceptual modeling language is not suited at all for modeling the module.

Since none of the above conceptual modeling languages are really sufficient for modeling VR at a high-level we decided to go for a new conceptual modeling approach (VR-WISE, which we will discuss in chapter 3) in order to make the design of a VR application more accessible for a non-VR specialist. This conceptual modeling approach has been designed with high-level modeling of VR in mind. However, as we will see throughout this dissertation, some modeling concepts from the above languages also appear more or less with the same meaning in our approach. Some examples of this are concept (representing a class), instance and role (as a concept or instance playing a certain role in a certain context). But the approach also contains a complete set of new modeling concepts specifically tailored towards VR such as connection relations to model physical connections between two objects.

2.3 Academic modeling approaches

Very little research on conceptual modeling for VR has been done. Care should be taken with the terms *modeling* and *high-level* in the context of VR. In the domain of VR, modeling is often used to indicate the process of shaping objects for virtual environments while high-level is often used to refer to the possibility to focus on what to render instead of how to render it. The term *conceptual modeling* as used in this dissertation denotes the modeling activity as described in section 2.2. With the term high-level we mean a high-level of abstraction from the implementation technology.

[22] states that the complexity of creating virtual environments rises from the fact that four modules need to be designed. As already indicated, these modules are form, function, behavior and interaction. In the context of this dissertation we are mainly concerned with form. In order to cope with the complexity [22] proposes a structural approach to developing VR applications called ASADAL/PROTO. Inside ASADAL/PROTO form is described using the Visual Object Specification (VOS). The primary purpose of VOS is to describe physical properties and configuration of physical entities. Spatial constraints can be used to define a structure that is similar to a scene graph (note that in VR a scene graph organizes a composite object by means of a tree). When a motion is applied to the parent, all the children are affected as well. However, there is no support in VOS to describe physical connections and constraints between different objects.

The lack of high-level design methodologies for VR was also addressed in [59] with the presentation of Virtual Reality Interface Design (VRID). Inside VRID, four key components are identified for designing VR interfaces. These are object graphics, objects behaviors, object interactions and object communications. VRID divides the design process into a high-level and a low-level design phase. The goal of the high-level design phase is to specify a design solution at a high level of abstraction. The output of the high-level design phase is a high-level representation of data elements and objects in the interface. The goal of the low-level design phase is to provide fine-grained details of the high-level representations and to provide details as to how objects will be represented. However, VRID does not provide a modeling language to support the high-level nor the low-level design phase.

Another research project which has similar goals as our approach is Ossa [56]. Ossa is a conceptual modeling system for virtual realities that uses conceptual graphs for representing knowledge and a production system to represent the world dynamics. However, no conceptual modeling constructs for modeling complex objects are present in Ossa.

More closely related work in the context of this dissertation can be found in the domain of (virtual) assembly modeling. Assembly modeling is a term used in Computer-Aided Design (CAD) to assemble several components together. Assembly modeling is an important part of product design since most products are assembled of several parts. Virtual assembly is based on the idea of performing the assembly operations in a virtual environment by directly manipulating parts.

We will now discuss a number of approaches inside the domain of assembly modeling which are related to the work in this dissertation.

2.3.1 The CODY Virtual Constructor

The CODY Virtual Constructor [66, 33, 34] is a system which enables an interactive simulation of assembly processes with construction kits in a virtual environment. The interaction during the assembly process happens either by means of direct manipulation or by means of natural language. In figure 2.5 the user interface of the CODY Virtual Constructor is illustrated. The example inside this figure is using a construction kit containing bolts, screws, . . . in order to assemble a toy airplane.
Connections happen by means of predefined points on the graphical objects. These predefined points are called connection ports. When a moved object is in a position so that one of its connection ports is close enough to the connection port of another object, a snapping mechanism will fit the objects together.
The core of the CODY architecture is based on a knowledge processing component that maintains two levels of knowledge, namely a geometric level and a conceptual level.

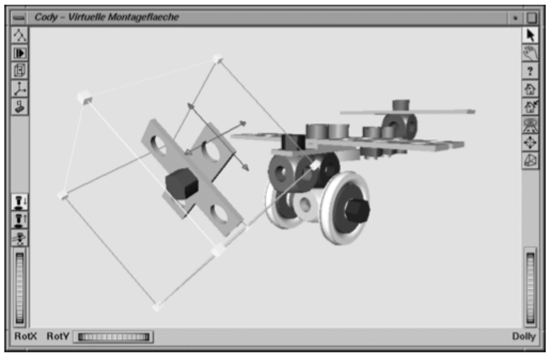


Figure 2.5: User Interface of the CODY Virtual Constructor [33]

- **Conceptual level:** The conceptual level contains conceptual knowledge about the mechanical objects of the construction kit used, such as their connection ports. Next to this static knowledge there is also a dynamically updated conceptual representation describing the current situation in the virtual environment.
- **Geometric level:** The geometric level contains generic knowledge about the mechanical objects' geometric properties such as the objects' wire frame, bounding box and center of gravity. The geometric level also contains some dynamic knowledge such as the current object positions and orientations.

For the representation of the knowledge inside the knowledge bases a frame-based representation language COAR (Concepts for Objects, Assemblies and Roles) has been developed. The following part of COAR language is an example of the specification for an Undercarriage assembly consisting of three parts, namely two *HALFAXLESYSTEM* objects and one *UNDERCARRIAGEBLOCK*.

```
Long-Term Concept:  UNDERCARRIAGE
    is-a:  ASSEMBLYGROUP
    part has-left-halfaxlesys #1:  HALFAXLESYSTEM
    part has-right-halfaxlesys #1:  HALFAXLESYSTEM
    part has-block #1:  UNDERCARRIAGEBLOCK
    pp-constraint connection ⟨has-block⟩ ⟨has-left-halfaxlesys⟩
    pp-constraint connection ⟨has-block⟩ ⟨has-right-halfaxlesys⟩
```

Discussion

Although the use of a virtual environment to perform the assembly process offers a more intuitive interaction, as does the possibility to perform the assembly process using natural language, the CODY Virtual Constructor approach has some disadvantages:

- The use of natural language offers a very intuitive way of describing an assembly. However, natural language is often ambiguous and incomplete. This means that the outcome of some natural language assembly modeling might not be what the designer wants which results often in a number of iterations before the desired result is reached.

- The COAR language was developed as an underlying representation mechanism for the knowledge bases. Hence the COAR language must be machine readable to allow the knowledge processing component of the virtual constructor to calculate some inferences. However, this conceptualization is not intuitive for non-VR or non-engineering people, as we have seen in the example.

- The CODY Virtual Constructor uses predefined points on the objects' geometry between which connections can be made. The use of these so-called connection ports has some disadvantages. Firstly the number of possibilities to make connections is limited. This way it offers less flexibility to the designer. The second disadvantage is that these connection ports must be defined in advance for the objects in a construction kit. A third disadvantage might occur with very large assemblies where it can be difficult for the designer to find his way through all the connection ports.

2.3.2 The Open Assembly Model

The aim of the Open Assembly Model (OAM) [49] is to provide a standard representation and exchange protocol for assembly information. In fact it is defined as an extension to the NIST Core Product Model (CPM) which was presented in [20] to which we refer the interested reader. Figure 2.6 shows the main schema of a part of the Open Assembly Model. The complete class diagram can be found in [49].

The class *Artifact* (which comes from the CPM) refers to a product or one of its components. It has two subclasses, namely *Assembly* and *Part*.

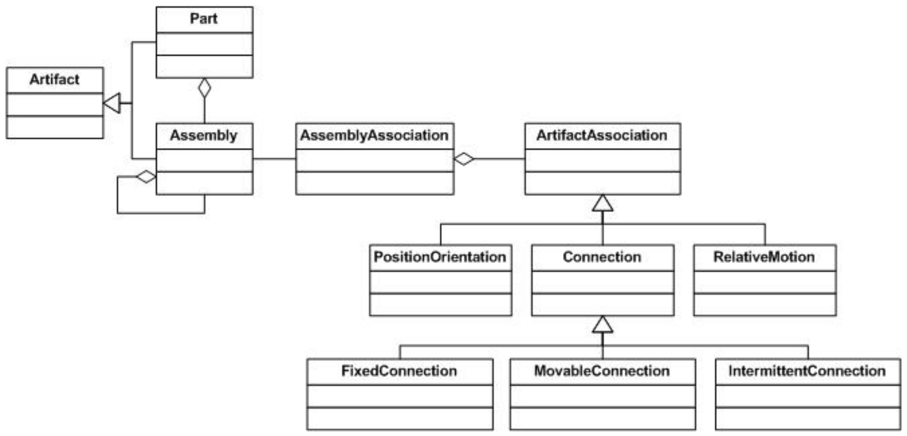


Figure 2.6: Partial class diagram of the Open Assembly Model

An *Assembly* is a composition of its subassemblies or parts. The class *AssemblyAssociation* contains information about assembly relationships. It is the aggregation of one or more *ArtifactAssociation* which is specialized into three other classes: *PositionOrientation*, *Connection* and *RelativeMotion*. *PositionOrientation* represents the relative position and orientation between two or more artifacts that are not physically connected. *RelativeMotion* represents the relative motions between two or more artifacts that are not physically connected. *Connection* represents a connection between artifacts. It is specialized into three classes: *FixedConnection*, *MovableConnection* and *IntermittentConnection*. Now, in order to illustrate the use of the Open Assembly Model we will look to a small example.

Example

Take for example a piston which needs to be placed inside a shaft. The piston can move inside the shaft. This example is illustrated in figure 2.7.

The instance diagram for the assembly relationships is presented in figure 2.8. We have an instance of *Assembly* called 'example'. Inside this assembly, there are two *Part* instances called 'piston' and 'shaft', connected to each other by means of a *MoveableConnection*. Note that the right side of the instance diagram was not discussed in figure 2.6. However, this represents an assembly feature association containing the information about the physics connection by means of a kinematic pair. This kinematic pair for the example is an instance of *RevolutePair* representing the information for

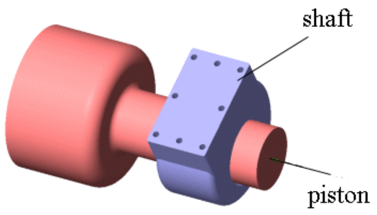


Figure 2.7: Example assembly consisting of a shaft and a piston

a revolute joint between the shaft and the piston.

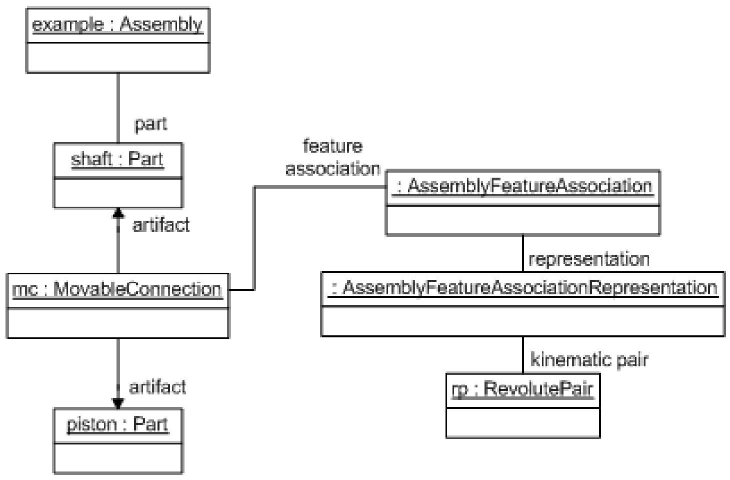


Figure 2.8: Instance diagram of an example assembly in OAM

Discussion

OAM has been designed to represent information used or generated inside CAD[3]-like tools. This way of representing the data can for example be used for transmitting the data between different development activities.

[3]Computer-Aided Design (CAD) uses computer-based tools that help engineers and other design professionals in their design activity.

This enhances the product development across different companies or even within one company. However, OAM is not targeting the modeling of assemblies on a conceptual level. It is targeting an underlying representation of assemblies inside the domain of engineering. Hence, it is note very usable as a conceptual modeling language for non-VR or non-engineering experts. This can be illustrated by means of the definition of kinematic constraints between two artifacts as we have seen in the example. In fact, these constraints come from the STEP standard (Standard for the Exchange of Product Model Data). STEP was designed to describe the physical and functional features of industrial product. STEP is only used late in the product development while OAM is claimed to be usable for representing information produced during the development process. The STEP representation for product data is very low-level and thus certainly not usable for non-engineering people. We refer the interested reader to [18] for more information on the STEP standard.
Now we get back to OAM. As already said, OAM uses the kinematic structure schema from part 105 (ISO 10303-105) of the STEP specification for representing kinematic constraints. This part defines the kinematic structure. Because of the high level of detail of kinematic constraints in the engineering domain, it is very hard to use. Since OAM makes use of this STEP part, this representation of kinematic constraints is not usable in a high-level representation of connections between parts. So in general we can conclude that OAM is not suitable for domain experts for modeling complex objects for a virtual environment on a high-level of description.

2.3.3 Virtual Assembly Design Environment

The Virtual Assembly Design Environment (VADE) [31, 32] is a VR-based engineering application that allows engineers to evaluate, analyze, and plan the assembly of mechanical systems. VADE has been developed at the Washington State University in collaboration with the National Institute for Standards and Technology (NIST).
The system utilizes an immersive virtual environment coupled with commercial CAD systems. VADE translates data from the CAD system to the virtual environment. Once the designer has designed the system inside a CAD system, VADE automatically exports the data into the virtual environment. Then, the VR user can perform the assembly. During the assembly process the virtual environment keeps a link with the CAD system. At the end of a VADE session, the design information from the virtual environment is made available in the CAD system.

Inside the virtual environment, designers can change some parameters using a 3D interface. The user can for example change the size of a part. Changes of parameter values are sent to the CAD system which can then regenerate the part.

However, the VADE system is intended for engineers and far from high-level. For example, kinematic motions during the assembly design are created from information coming from the CAD system. CAD systems are only accessible to experts in CAD modeling. VADE is therefore not usable as a high-level modeling environment for complex objects in a virtual environment.

2.3.4 Multi-user Intuitive Virtual Environment

The Multi-user Intuitive Virtual Environment (MIVE) [54, 24, 57] provides a simple way for objects to constrain to each other without having to use a complete constraint solver, as is usually the case for constraint-based modeling. Inside MIVE, there are a number of interesting constraint ideas. First there are the virtual constraints. Each object in the scene is given predefined constraint areas. These areas can then be used to define so-called offer areas and binding areas. Offer areas define locations on an object where other objects may reside while binding areas define locations on an object that can be positioned inside certain offer areas. Note that in this dissertation we will also introduce a high-level modeling concept to allow modeling this kind of constraint. This modeling concept was inspired by the work presented in MIVE. Next to virtual constraints, MIVE also contains so-called negative constraints which can be used for preventing certain objects to reside in certain spaces of the virtual environment. This constraint can for example be used to prevent that objects larger than a door of a room are placed in front of that door.

However, one major disadvantage of the MIVE approach is that for example for the virtual constraints each object in the scene needs predefined areas. Therefore it is difficult to reuse existing virtual objects without adapting them to the MIVE approach.

2.4 X3D

X3D [1] (eXtensible 3D) is the ISO standard for real-time 3D computer graphics that can be seen as the successor of VRML. Although it is not a real academic approach it is also not commercial. It is a royalty-free open standard file format. Therefore we introduce it here. X3D makes use of a scene graph structure containing all the objects in the system and their relationships by means of nodes. Such a scene graph is encoded in X3D using an XML-syntax. The interpretation, execution and presentation of such X3D-files is done by a browser. Recently, a revision of the X3D architecture and base components specification includes a rigid body physics component (clause 37 of part 1 of the X3D specification). This part describes how to model rigid bodies and their interactions by means of applying basic physics principles to effect motion. It offers various forms of joints that can be used to connect bodies together and allow one body's motion to effect another. Examples of joints offered are BallJoint, SingleAxisHingeJoint, SliderJoint or UniversalJoint. All these joints are offered as an X3D node that can be used in the specification of the scene graph. We will now look to the SingeAxisHingeJoint node representing a joint with a singe axis around which the connected bodies may rotate:

```
SingleAxisHingeJoint :  X3DRigidJointNode {
    SFVec3f  [in,out]   anchorPoint         0 0 0
    SFVec3f  [in,out]   axis                0 0 0
    SFNode   [in,out]   body1               NULL [RigidBody]
    SFNode   [in,out]   body2               NULL [RigidBody]
    SFFloat  [in,out]   maxAngle            π
    SFNode   [in,out]   metadata            NULL [X3DMetadataObject]
    SFFloat  [in,out]   minAngle            -π
    MFString [in,out]   mustOutput          "NONE" ["ALL","NONE",...]
    SFFloat  [in,out]   stopBounce          0 [0,1]
    SFFloat  [in,out]   stopErrorCorrection 0.8 [0,1]
    SFFloat  [out]      angle
    SFFloat  [out]      angleRate
    SFVec3f  [out]      body1AnchorPoint
    SFVec3f  [out]      body2AnchorPoint
}
```

The minAngle and maxAngle fields can be used to specify the angles which the connected objects are allowed to rotate around the axis specified by means of the anchorPoint and axis fields. The body1 and body2 fields specify the bodies that are connected by means of the joint. These are all examples of input fields which specify properties of the joint node. The angle field is an example of an output field reporting the current relative angle between the two connected bodies.

Discussion
First note that X3D is sometimes entitled as being high-level. However, high-level in the context of X3D means that the focus lies of what to render in a scene instead of how to render the scene. Although recently the possibility to specify complex objects by means of different joint types connecting two bodies has been added, the way to specify these constraints is far from high-level. As we illustrated this with the SingeAxisHingeJoint. It is not intuitive for a non-VR expert to specify a hinge constraint by means of points and vectors. Furthermore, reasing over X3D specifications is far from easy.

2.5 Commercial modeling approaches

2.5.1 SimMechanics

SimMechanics is a set of block libraries and special simulation features to be used in the Simulink environment[4]. Simulink is a platform for simulation in different domains and model-based design for dynamic systems. Simulink provides an interactive graphical environment that can be used for building models. These models are build by drag-and-drop of blocks coming from a customizable set of block libraries.

The SimMechanics block libraries contain the elements for modeling mechanical systems consisting of a number of rigid bodies connected by means of joints representing the translational and rotational degrees of freedom of the bodies relative to one another. The libraries of SimMechanics which are closely related to the research described in this dissertation are the *Bodies Library* and the *Joints Library*.

Bodies library

The Bodies library provides the Body block for representing user-defined bodies by their mass properties, their positions and orientations and their attached Body coordinate systems. Body coordinate systems (CS) are fixed in the body and move with it. Each body has at least one CS, namely at it's center of gravity. Additional CSs can be specified for connecting joints as we will see later. Figure 2.9 shows the graphical representation of a body in SimMechanics. As we can see, there is one additional CS defined for this body, namely CS1, which will be used for connecting a joint.

[4]http://www.mathworks.com/products/simulink

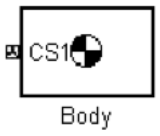


Figure 2.9: Graphical representation of a rigid body in SimMechanics

SimMechanics has two different tools for visualizing bodies. The first tool is a Matlab Handle Graphics-based visualization tool built into SimMechanics. The second tool is an optional visualization tool based on the Virtual Reality Toolbox[5]. However, both tools represent the bodies in an abstract simplified form, namely by means of convex hulls or as equivalent ellipsoids.

Joints library

Next, the Joints library provides the blocks to represent the relative motions between bodies. Note that SimMechanics bodies unlike physical bodies do not have degrees of freedom (DoF). A joint in SimMechanics represents the DoFs that one body (called the follower) has relative to another body (called the base). Therefore SimMechanic joints only add degrees of freedom to a body unlike physical joints which may both add or remove DoFs to a body.

SimMechanics provides a Joints library that can be used for modeling various types of joints. Each joint block represents one or more joint primitives that together specify the degrees of freedom that a follower has relative to the base. An overview of the joint primitives is given in table 2.2.

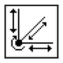
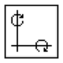
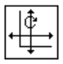
Next, composite joints are blocks containing a combination of joint primitives which enable the user to specify multiple rotational and translational degrees of freedom of one body relative to another. Table 2.3 gives a couple of examples of composite joint blocks offered in SimMechanics.
So relative motion of bodies with respect to one another is represented by connecting their body blocks with a joint block. A joint block is always connected to a specific point on a body. This specific point is specified by means of a body coordinate system. An example of a revolute joint connecting two bodies is shown in figure 2.10. In this example, the revolute

[5]http://www.mathworks.com/products/virtualreality

Table 2.2: Joint primitives in SimMechanics

Joint name	Notation	Description
prismatic	↕	A prismatic joint block represents a singe translational DoF along a specified degree of translational freedom (thus along a specific axis between two bodies).
revolute	¢	A revolute joint block represents a single rotational DoF around a specified axis between two bodies.
spherical	●	A spherical joint block represents three rotational DoFs at a singe pivot point. This type of joint is also known as a ball-in-socket joint.
weld	⊢▫⊣	A weld joint block represents a joint with no DoFs. Two bodies connected by means of a weld joint have no possible relative motion.

Table 2.3: Joint composites in SimMechanics

Joint name	Notation	Description
Six-DoF		A Six-DoF joint block represents a composite joint with three translational DoFs (as three prismatic primitives) and three rotational DoFs (as one spherical primitive).
Universal		A universal joint block represents a composite joint with two rotational degrees of freedom (as two revolute primitives).
In-plane		An in-plane joint block represents a composite joint with two translational degrees of freedom (as two prismatic primitives).

joint connects two bodies. Inside the revolute joint block a B and F indicate which body is the base and which body is the follower. The joint is connected to a specific point on both bodies defined as CS1 for each body. This CS1 is a body coordinate system. The axis for the revolute joint is mathematically defined inside a graphical user interface offered for specifying the details of the revolute joint.

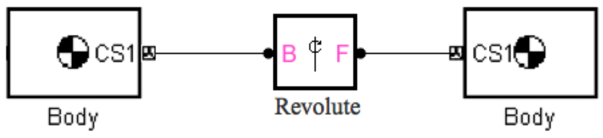


Figure 2.10: Example of the use of a revolute joint in SimMechanics

Discussion

Although SimMechanics is already on a higher-level of abstraction (than for example physics engine programming), it is still too low-level to be general usable for domain experts from different domains. After all, SimMechanics has been developed to model mechanical systems. One example of something which may not be very intuitive to some domain experts is the fact that bodies do not have any degrees of freedom in contrast to physical objects which have six degrees of freedom. For real-world physical objects adding joints means restricting the object's degrees of freedom while in SimMechanics adding a joint means adding degrees of freedom to a body. Another example is that the details of a joint block need to be specified in a mathematical way. Another disadvantage of the approach is that there is no possibility to do some intelligent reasoning.

2.5.2 MotionWorks For SolidWorks

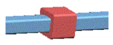
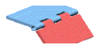
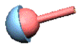
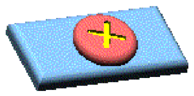
SolidWorks[6] is a 3D computer-aided design (CAD) program in which 3D parts can be created. These 3D parts are made by using several features. Features can be for example shapes and operations like chamfers or fillets. Most of the features are created from a 2D sketch. This is a cut-through of the object which can for example be extruded for creating the shape feature.

MotionWorks makes it possible to define mechanical joints between the parts of an assembly inside SolidWorks. It is fully integrated into SolidWorks. MotionWorks also offers the possibility to simulate the dynamic behavior of a model, analyze the results and refine the design. To simulate the dynamic motion in an assembly, mechanical joints need to be defined between the parts of the assembly. In MotionWorks this can be done by selecting the required joint type from a list. MotionWorks contains different families of joints. The type of joints which are closely related to the work presented in this dissertation are the *permanent joints*. These joints are based on different combinations of rotating and translating degrees of freedom. Some examples of these permanent joints are given in table 2.4. In this table their name as well as their representation in the graphical user interface is given. A complete overview of all permanent joints can be found in [2].

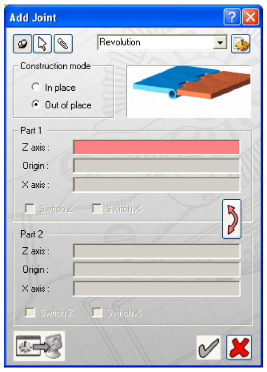
When creating a joint between two parts, the user can choose for the 'in place' or the 'out of place' option. 'In place' means that the parts are already in the desired orientation while 'out of place' means that the parts to connect by means of the joint are not in the desired orientation. Note that, as we will see later, in the approach proposed in this dissertation, orientations are

[6]http://www.solidworks.com

Table 2.4: Example permanent joints in MotionWorks

Joint name	Representation
Prismatic	
Revolution	
Spherical	
Planar	

automatically calculated according to the type of connection between two parts. When defining joints, MotionWorks automatically creates coordinate frames on each part. However the alignment of the z-axis is important since this is the primary axis about which a translation or rotation will happen. Figure 2.11 illustrates the graphical user interface for creating a revolution joint between two parts.



Add Joint

Revolution

Construction mode
In place [✓ Out of place]

Part 1
Z axis :
Origin :
X axis :
☐ Switch Z ☐ Switch X

Part 2
Z axis :
Origin :
X axis :
☐ Switch Z ☐ Switch X

Figure 2.11: GUI for the creation of a revolution joint in MotionWorks

Next to creating the joint between two parts, all parameters like initial position, torques, friction, bounds, . . . need to be defined.

Discussion

SolidWorks (and thus also MotionWorks) is a pure CAD modeling tool. These kinds of tools are targeting design professionals such as engineers or architects. SolidWorks is very powerful in modeling mechanical systems while MotionWorks is the solution for kinematic and dynamic simulation of moving assemblies. Nevertheless, they are not suited for non-experts in the domain of CAD. This is illustrated by means of the vocabulary they are using to specify certain parameters of parts or joints.

2.5.3 3D Studio Max

3D Studio Max (3ds max) is a 3D modeling software in the category of authoring tools. Note that there are other tools available which are more or less similar to 3ds max. Some examples are Blender[51], Maya[61], . . . Also note that this kind of tools is using a low-level VR approach to modeling virtual environments. However, since they also offer a number of possibilities to model complex objects, we will discuss 3ds max here for the completeness of this related work overview.

3ds max provides grouping features which enable the user to organize all the objects with which he is dealing. This makes it easier to select the grouped objects or to transform them. Groups can also be grouped creating nested groups.
3ds max provides also another way of organizing objects, namely by building a linked hierarchy. A child object is an object that is linked to and controlled by a parent. Once two objects are linked all transformations applied to the parent are equally applied to its children. Next 3ds max provides a number of constraints that can be used to force objects to stay attached to another object. Actually the designer can use these constraints to restrict the motion of an object. 3ds max provides the following constraints: Attachement, Surface, Path, Position, Link, LookAt and Orientation. The Attachement constraint can for example be used to determine an object's position by attaching it to the face of another object. Another example is the LookAt constraint that can be used to force an object so that it is always oriented towards a target object. All these constraints are explained in more detail in [41]

Discussion

Although authoring tools are intended to create virtual environments without the need for detailed programing one needs to be an expert in the domain of VR to be able to use an authoring tool like 3ds max. The vocabulary used in the menus and dialogs of such an authoring system is very domain specific. Terms like NURBS, splines or morph are more or less meaningless for a native user. There is no way to describe the virtual environment using the terminology of the application domain.

2.6 Summary

In this chapter we have given an overview of the relevant background and related work. We gave a brief introduction to Virtual Reality. Next we also discussed conceptual modeling and reviewed three well-known conceptual modeling languages and we discussed their usability for high-level modeling in the domain of VR.
We discussed two groups of related work. First we discussed the academic approaches. More specifically we discussed some approaches from the domain of (virtual) assembly modeling. Next we discussed a number of well-known commercial tools.

Chapter 3

Overview of the VR-WISE approach

In this chapter an overview of the VR-WISE approach will be given. The research described in this dissertation is performed in the context of this approach. This chapter is structured as follows. First we will give a general introduction to the VR-WISE approach (section 3.1). Next the different steps of the VR-WISE approach are discussed in section 3.2. Finally we will discuss the high-level modeling concepts used within the VR-WISE approach. Subsection 3.3.1 gives an overview of the modeling concepts introduced for modeling simple objects while subsection 3.3.2 briefly discusses the modeling of behavior inside the VR-WISE approach. We will finish this chapter with some conclusions.

3.1 Introduction

The goal of the VR-WISE approach is to facilitate and shorten the development process of Virtual Environments (VE) by introducing an explicit conceptual design phase in the development life cycle of a VR-application. During this conceptual design phase conceptual specifications (also called conceptual models) are created. Such a conceptual specification is a high-level representation of the objects inside the VE, the relations that hold between these objects and how these objects behave and interact with each other and with the user. A conceptual specification is free from any implementation details. The VR-WISE approach offers a set of intuitive modeling concepts in order to build these conceptual specifications.

The conceptual specifications are internally represented by means of ontologies. In its most simple form, an ontology can be seen as an abstraction of a computer-based lexicon, thesaurus, glossary or some other type of structured vocabulary, suitably extended with knowledge about a given domain [26][25]. Using ontologies as the underlying representation formalism for the conceptual specifications means that the modeling concepts and all the information collected during the design phase are maintained in ontologies. The use of ontologies offers some advantages, e.g., they can be used for intelligent reasoning, as we will see later on. However, it is not necessary to rely on ontologies as the underlying representation formalism for the conceptual specifications. In principle other representation formalisms may also be suitable. One example of another possible representation formalism is Frame-Logic (F-Logic). As we will see in part II of this dissertation, we can formalize the VR-WISE modeling concepts presented in this dissertation by means of F-Logic

The use of a conceptual level will improve reusability, extensibility and modularity of the design. Other advantages of the VR-WISE approach are:

- *The VR-WISE approach enhances the participation of domain experts into the development of a VR-application.* The fact that the design of the virtual environment is expressed in terms of the domain, for which the virtual environment is created (application domain), makes the design more intuitive for a domain expert. For instance, specifying a car design in terms of wheels, a hood, a trunk, . . . can be more intuitive to a domain expert than having to deal with VR primitives such as boxes and cylinders.

- *The VR-WISE approach offers a (semi-)automatic generation of the virtual environment.* The conceptual specifications provide a considerable amount of information which allows an easier (semi-) automatic generation of the VR application. A proof-of-concept implementation (see chapter 12) shows that it is possible to transform most, if not all, high-level modeling concepts into low-level implementation details for the actual virtual environment.

- *The conceptual specifications can be used as a basis for discussing the design of a virtual environment among the different stakeholders of a project.* Current practice when designing a VR application are brainstorm sessions, which are held with the different stakeholders in the project. Based on these meetings, notes and sketches are made. However, natural language and sketches are informal, ambiguous and very often incomplete. Mistakes can be made because words can be interpreted in a variety of ways. The consequence of such misunderstandings is that (a part of) the VR application may not satisfy the expectations of the stakeholders and adaptations or even rebuilding the application may be needed. Another disadvantage of natural language and sketches is that they cannot be processed easily automatically in order to generate the source code for the VR application. The conceptual specifications created using the VR-WISE approach can serve as a formal communication basis between the domain experts, the VR-experts and other stakeholders in the project.

- *The VR-WISE approach allows intelligent reasoning.* Starting from a scene graph or APIs such as Java3D [11] it is not easy to perform some intelligent reasoning. However, intelligent reasoning becomes easier when using mechanisms such as ontologies or F-Logic as the underlying formalism for the specifications. For example when using F-Logic as the underlying representation mechanism we can use reasoning tools such as Flora-2 [68] or OntoBroker [23] to derive extra information about the virtual environment.

Note that the VR-WISE approach is a general approach for VR development. The approach is not especially dedicated to a particular application domain. However the models used inside the VR-WISE approach can be seen as abstract prototypes that could be refined later on for specific domains.

It is also possible to further refine the general approach of VR-WISE towards specific application domains. We have experimented by customizing the VR-WISE approach for the development of VR-shops. This means that we have pre-defined a number of object types and behaviors and provision for plugging-in shopping functionality (such as buying or reserving products). More details about this experiment can be found in [63].

3.2 The three steps of the VR-WISE approach

The design process used in the VR-WISE approach is divided into three steps. These steps are mainly sequentially. Figure 3.1 illustrates these steps which are the *specification step*, the *mapping step* and the *generation step.*

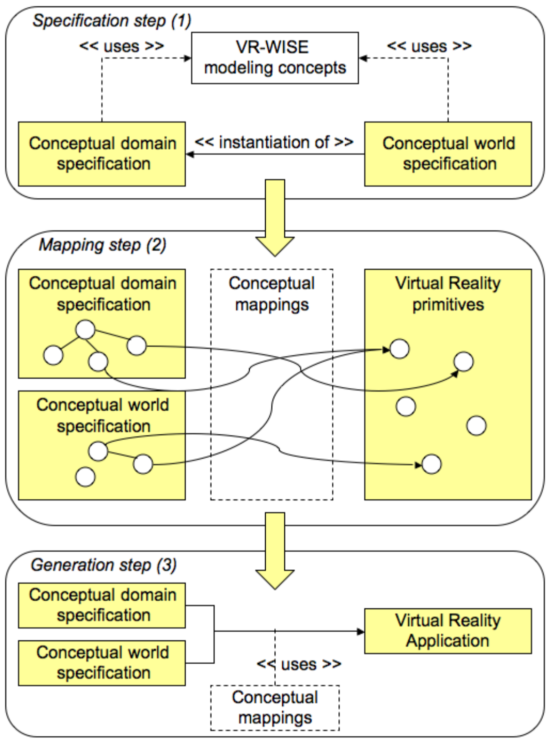

Specification step (1)
<< uses >>
VR-WISE modeling concepts
<< uses >>
Conceptual domain specification
<< instantiation of >>
Conceptual world specification
Mapping step (2)
Conceptual domain specification
Conceptual mappings
Virtual Reality primitives
Conceptual world specification
Generation step (3)
Conceptual domain specification
Virtual Reality Application
Conceptual world specification
<< uses >>
Conceptual mappings

Figure 3.1: Overview of the VR-WISE approach

3.2.1 The specification step

During the specification step the designer can specify the virtual environment at a high-level using the intuitive modeling concepts offered by the VR-WISE approach. The specification has two levels since the VR-WISE approach follows to some degree the object-oriented paradigm [30]. The first level is the *domain specification*. The domain specification describes the concepts of the application domain (comparable to object types in OO-design methods). It also describes possible relations that may hold between these concepts. In the domain of vehicles for example, the domain specification could contain concepts such as car, bike, wheel or boat and relations such as "a bike has two wheels". The second level of the specification is the *world specification*. The world specification contains the actual conceptual description of the virtual environment to be built. This level of specification

is created by instantiating the concepts specified in the domain specification. These instances actually represent the objects that will populate the virtual environment. In the vehicle example, there can be multiple car-instances, multiple bike-instances and multiple boat-instances. Next to concept- and instance specific information (attributes), such as size, color and location, semantic information which will not directly be represented in the Virtual Environment can be modeled. In the vehicle example this could be the price of a specific car- or boat-instance. To specify information about concepts and instances, their properties and relationships, a number of high-level modeling concepts are provided.
Note that in the rest of this dissertation we will use the following convention for the use of the terms concept, instance, and object:

- **Modeling concept:** We will use the term "modeling concept" when we mean a high-level modeling concept provided by the VR-WISE approach to create the conceptual specifications. A spatial relation (which we will see later) is an example of a modeling concepts.
- **Concept:** "Concept" is the term that will be used when we mean a concept in the conceptual domain specification. We could for example model the concept *car*.
- **Instance:** The term "instance" will be used for instances of concepts which appear in the conceptual world specification. An instance of the concept *car* could be the instance *Mercedes-A*.
- **Object:** We will use the term "object" for a concept as well as for an instance and when it is not necessary to make the distinction.
- **VR object:** Finally the term "VR object" will be used when we want to refer to the actual 3D object in the virtual environment.

3.2.2 The mapping step

During the mapping step the conceptual specifications are mapped onto the implementation level. Similar as for the specification step, we have two levels. On the first level we have the mappings from the concepts in the domain specification towards VR objects. The purpose of these mappings is to specify defaults for the representation of instances of concepts in the virtual environment. For our example domain, the domain of vehicles, we could map the concept wheel to a VR object cylinder. Note that this is a very simple and easy one-to-one mapping. However, more complex mappings are possible as well. Although there may be several instances of the same concept, they may in some cases require a different representation in the virtual environment. Therefore on the second level we have the mappings from the instances in the world specification onto VR objects. This level of mapping allows the designer to override the default mappings specified for the concepts in the domain specification. For example, there may be an instance of a wheel that need to be represented as a sphere instead of the default cylinder representation.

3.2.3 The generation step

The last step is the generation step. During this step the actual source code for the virtual environment, which was specified in the domain and world specification, is generated. This means that the conceptual specifications are converted into a working application by means of the mappings given during the mapping step. In principle, different VR languages can be supported. We will discuss this further when we discuss the implementation of the VR-WISE approach in chapter 12.

This section has presented the general structure of the VR-WISE approach. In the next section we will present the high-level modeling concepts provided by the VR-WISE approach.

3.3 High-level modeling concepts

In this section we will give an overview of the high-level modeling concepts that can be used inside the VR-WISE approach in order to build the conceptual specifications. For each high-level modeling concept we will provide an informal definition and a graphical notation. Formal definitions will be given in chapter 7. We can categorize the modeling concepts in three categories. The first category contains the modeling concepts for modeling simple VR objects. The second category contains the modeling concepts for specifying complex VR objects, which are assemblies of simple and/or complex objects. This category is the subject of this dissertation. Therefore in chapter 4, we will discuss these modeling concepts into more detail. The third category of modeling concepts consists of those that can be used to model the behavior of VR objects. This category is outside the scope of this thesis and will be the subject of another PhD-thesis. Hence we will only briefly introduce this category of modeling concepts here.

3.3.1 Modeling concepts for simple objects

Concepts

Object types are represented as concepts in the VR-WISE approach. A concept is comparable to a class in object-oriented design [30]. Concepts are used in the domain specification. They can have two types of properties: visual properties and non-visual properties.

- *Visual properties*
  Visual properties will be reflected in the Virtual World. Examples of visual properties are color, width, depth, height, etc.

- *Non-visual properties*
  Non-visual properties are not reflected in the Virtual World, such as mass, name, price, etc. This type of properties can also be used to represent some semantic information (real world information) about the object. Suppose we are modeling a virtual shop, this type of properties can be used to model the price of a product, its availability, its delivery cost, etc.

Definition 3.3.1 gives an informal definition of the modeling concept ”concept”.

Definition 3.3.1 ***(Concept)***
A concept represents an object type from the application domain which is relevant for the VR-application. A concept can have a number of visual as well as non-visual properties.

Graphical notation

A concept is represented as a rectangle containing the name of the concept. This graphical notation is illustrated in figure 3.2.

ConceptName

Figure 3.2: Graphical notation for the modeling concept *Concept*

Using the graphical development environment developed for VR-WISE, properties for concepts, visual as well as non-visual can be specified. Also default values can be specified for the properties

Instances

Concepts are specified at the domain specification level. Next the designer needs to instantiate these concepts in order to populate the virtual environment. An instance of a concept inherits all properties; visual as well as non-visual, from the concept it is an instance of. However, all these properties' values can be overwritten at the instance level. Instances are specified in the world specification. Definition 3.3.2 gives an informal definition of the modeling concept "instance".

Definition 3.3.2 ***(Instance)***
An instance is an instantiation of a concept. It inherits all properties of the concept it is a instance of. An instance represents an actual VR object that will populate the VE.

Graphical notation

An instance is graphically represented as an ellipse. Inside the ellipse, a tag indicates the instance name and the name of the concept it is an instance of. The tag is constructed as *ConceptName:InstanceName*. The graphical notation for instances is illustrated in figure 3.3.

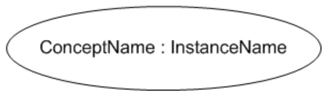


Figure 3.3: Graphical notation for the modeling concept *Instance*

Spatial Relations

Spatial relations allow the designer to position instances in the virtual world in an intuitive way. Spatial relations can be used to position instances relative to other instances. Instead of having to position each instance at concrete coordinates in the virtual world, the designer can, for example, specify that a specific instance is positioned at a certain distance left of another instance. Spatial relations can be used in the domain specification as well as in the world specification. Therefore, we will use the term object when dealing with spatial relations. In the domain specification, the spatial relations are used to specify default positions for the instances of a concept. Definition 3.3.3 gives the informal definition of a spatial relation.

Definition 3.3.3 ***(Spatial relation)***
A spatial relation specifies the position of an object relative to some other object in terms of a direction and distance.

Graphical notation

A spatial relation is graphically represented by a rounded rectangle (the symbol for a relation) containing an icon indicating that the relation is a spatial one. Below this icon the actual information for the spatial relation is specified: the direction and the distance. The graphical notation is illustrated in figure 3.4.

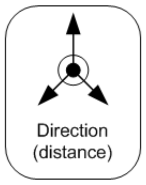


Figure 3.4: Graphical notation for the *spatial relation*

For specifying the spatial relation the following directions may be used: left, right, front, back, top and bottom. These directions may be combined. However, not all combinations make sense. Take A, B and C to be sets of simple directions defined as follows:

A = {left, right}
B = {front, back}
C = {top, bottom}

A combined direction exists of minimal two and maximal three simple directions. However, there may be only one direction of each of the sets A, B and C in the combined direction. For example, we may use the combined direction "left top" but we may not use "left right" since left and right belong to the same set.

Example

Suppose we have two instances, myCar and myHouse, which are instances of the concepts Car and House respectively. Suppose we want specify that myCar is 3 meters in front of myHouse. This can be specified as illustrated in figure 3.5.

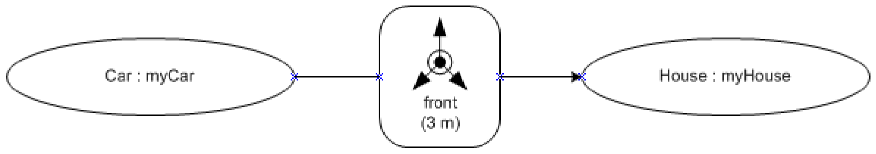


Figure 3.5: Example of the use of spatial relations

Note that the spatial relation symbol is connected to both objects by means of an arrow. In this case, the direction of the arrow indicates that the instance myCar is in front of the instance myHouse and not vice versa. The direction of the arrow actually indicates the reading direction: myCar is in front of myHouse.

Orientation Relations

In VE's it is also necessary to orient objects. Inside VR-WISE, objects have two types of orientations. An object has an internal orientation and an external orientation.

- **internal orientation:** The internal orientation of an object can be used to specify which side of the object is defined as the front, back, left, right, etc. An internal orientation is actually defined by a rotation of the local reference frame[1] of the object around some of the axes of the global reference frame[2] (which is equal to the default reference frame). This principle is illustrated in figure 3.6. Figure 3.6(a) shows the default internal orientation of an object. Next, in figure 3.6(b) an internal orientation of 45 degrees counterclockwise around the front direction axis is illustrated. As we can see, only the local reference frame of the object is rotated while the object itself maintains its orientation inside the virtual world. Actually we changed the left-right and top-bottom sides of the object.

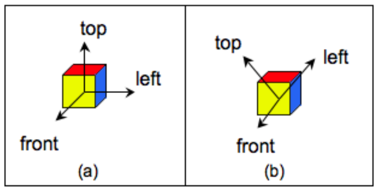


Figure 3.6: (a) default orientation; (b) internal orientation 45 degrees counterclockwise around the front axis

- **external orientation:** The external orientation of an object can be used to rotate the object itself around some of the axes of its local reference frame. This means that an object will be rotated around some of the axes of its reference frame and this will be visible in the virtual environment. The external orientation is illustrated in figure 3.7. Figure 3.7(a) shows the default orientation of an object while figure 3.7(b) shows an external orientation using a rotation of 45 degrees counterclockwise around the front axis. As we can see, the complete concept has been rotated.

In the VR-WISE approach, the internal orientation of a concept or instance can be changed by means of its visual properties. To change the external

[1]Local reference frame refers to the coordinate system local to the object. With other words, the coordinate system where the origin equals the origin of the object.

[2]Global reference frame refers to the coordinate system of the virtual environment. The origin of the global reference frame is equal to the origin of the virtual environment.

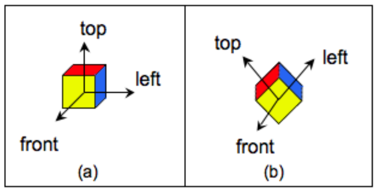


Figure 3.7: (a) default orientation; (b) external orientation 45 degrees counterclockwise around the front axis

orientation we developed two types of orientation relations.
The first type of orientation relation is used to specify the orientation of an object relative to another object. Using this type of orientation relation the designer can for example specify that an object is oriented with it's front side towards another object it's left side. This type of orientation relation is called *orientation by side*. Definition 3.3.4 gives an informal definition of the orientation by side relation.

Definition 3.3.4 ***(Orientation by side relation)***
An orientation by side relation specifies how some object is oriented towards another object. It can be used to specify which side of an object is oriented to which side of another object.

Graphical notation

The *orientation by side* relation symbol is a rounded rectangle. The relation has two parts; each needs to be connected to a concept or instance. Inside such a part the designer specifies which side of the object (concept or instance) needs to be oriented towards the other object. Therefore he may use the sides left, right, top, bottom, front and back or a combination of them. The combination has to follow the same rule as the combined direction we have seen with the spatial relations.
The graphical symbol for the orientation by side relation also contains a specific icon. The graphical notation for the orientation by side relation is illustrated in figure 3.8.
Example

Again suppose we have two instances, myCar and myHouse as defined earlier. Now suppose the designer wants to specify the fact that myCar is

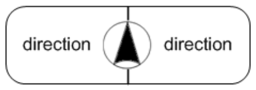


Figure 3.8: Graphical notation for the *orientation by side relation*

oriented with its right side towards the front side of the house. The specification for this fact is shown in figure 3.9. Note the use of the arrow. The arrow indicates that the orientation of the instance *myCar*, and not the orientation of the instance *myHouse*, needs be changed to fit the requirements of the orientation relation.

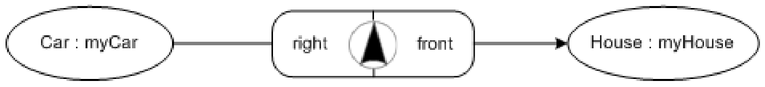


Figure 3.9: Example of the use of the orientation by side relation

The second type of orientation relation is the *orientation by angle* relation. This relation can be used to orient a single object by rotating it around an axis of its local reference frame over a certain angle. Definition 3.3.5 gives the informal definition of the orientation by angle relation.

Definition 3.3.5 ***(Orientation by angle relation)***
An orientation by angle relation specifies a rotation of an object over a certain angle around some specified axis.

Graphical notation

The *orientation by angle* relation symbol is a rounded rectangle containing the icon specific for this type of relation. Inside the rounded rectangle the designer can specify the axis around which the rotation needs to be done and the angle of the rotation. The axes that can be used are front-to-back, left-to-right and top-to-bottom. This graphical notation is illustrated in figure 3.10.
Example

Suppose we have an instance myCar, as defined earlier. If the designer

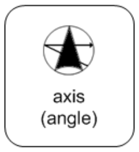


Figure 3.10: Graphical notation for the *orientation by angle* relation

wants to specify that the instance myCar needs to be rotated 45 degrees around its top-to-bottom axis he can use the orientation by angle relation for this. This example is shown in figure 3.11.

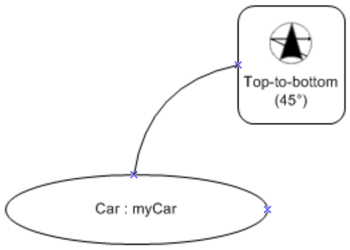


Figure 3.11: Example of the use of the orientation by angle example

So far, we have introduced high-level modeling concepts for specifying simple objects and their position and orientation inside the virtual environment. In the next section we will briefly describe the modeling of behavior inside the VR-WISE approach.

3.3.2 Modeling concepts for behavior

The VR-WISE approach also provides high-level modeling concepts for specifying behavior. The approach used for specifying behavior in the VR-WISE approach is called *action-oriented*. This means that the focus is on the actions an object needs to perform rather than on the states an object can be in. The behavior can be specified independent from the structure of the objects and from the interaction used to invoke the behavior. This improves the reusability of the behavior specifications.
The behavior specification process contains two steps, the *behavior definition specification* and the *behavior invocation specification*.

- The *behavior definition specification* allows the designer to define different behaviors for an object.

- A *behavior invocation specification* is created for each behavior definition specification. It assigns the behaviors defined in a behavior definition specification to an actual object and denotes the events that may trigger the behavior.

In the VR-WISE approach a set of high-level modeling concepts is developed which can be used to create behavior definition specifications and behavior invocation specifications. These modeling concepts are the topic of another PhD. We refer the interested reader to [46], [47], [45] and [48].

However, it is interesting to have a closer look to the relation between connections and constraint for complex objects and behavior. Suppose we have two concepts, a *Base* which is connected to a *Rail* by means of a slider constraint. This means that the base may only slide along the rail. Now we will look to behavior which is defined for the base concept. Figure 3.12 shows an example of a behavior definition diagram. This diagram defines an actor, which is an abstract concept, called *Primary*. This actor has the behavior called *MoveIt*. The behavior *MoveIt* is defined as a move towards the left over 3 meters.

Figure 3.13 gives an example of a behavior invocation diagram. This diagram couples the behavior that we have specified in figure 3.12 to the *Base* concept. So, the diagram in figure 3.13 actually models that when the *Base* concept is touched the behavior *MoveIt* is performed. The *Base* concept acts as the actor *Performer*.
Now suppose we defined the slider constraint between the *Base* and the *Rail* concept in such a way that the *Base* is only allowed to move left over

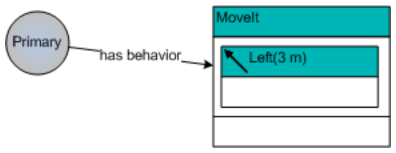


Figure 3.12: Behavior Definition Diagram example

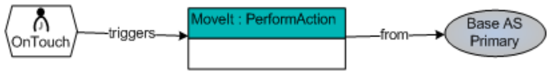


Figure 3.13: Behavior Invocation Diagram example

2 meters along the rail. When the *Base* concept is touched, the *MoveIt* behavior will try to move the *Base* 3 meters to the left. However, this behavior will be limited by the slider constraint because this constraint only allows it to move 2 meters to the left. Hence, we can say that connection relations and constraints will limit the behavior of objects connected to each other.

3.4 Use of the VR-WISE approach

Note that the VR-WISE approach is a first step towards a complete design methodology for virtual environments. Such a design methodology contains a number of phases such as a requirements phase, a design phase, an implementation phase, etc. A number of research projects investigated the need for a design methodology for VE's. For example, [53] introduces the Concurrent and LEvel by Level Development of VR Systems (CLEVR). [19] gives an overview of known work in this field and also proposes a design methodology for VE's. Future research may be performed in order to investigate if we can integrate the VR-WISE approach into such an existing design methodology or to investigate the creation of a new design methodology of which the VR-WISE approach will be part.

Furthermore, some modeling languages also contain a step-by-step design procedure which can be used to create the conceptual model. For example, ORM [27, 28] contains such a 7-step design procedure. For the VR-WISE approach we may also think of such a design procedure in order to create a conceptual model using the high-level modeling concepts. *"Model simple* *objects"*, *"Model complex objects"*, *"Model behavior"*, ... are examples of steps that can be needed to come to a complete conceptual model. However, further research is needed for the creation of such a step-by-step design procedure for the VR-WISE approach.

3.5 Conclusion

In this chapter we have given an overview of the VR-WISE approach. The goal of the VR-WISE approach is to facilitate and shorten the development process of virtual environments. This is done by introducing a conceptual design phase in which high-level modeling concepts are used to create conceptual specifications. The introduction of a conceptual design phase and the use of high-level modeling concepts enhances the participation of domain experts into the development of a VR-application. Results of this research have been published in [62], [9], [10] and [43].

As we have seen in this chapter, there are mainly three categories of high-level modeling concepts. The first category contains the modeling concepts that can be used for specifying simple VR objects. The second category contains the high-level modeling concepts that can be used for specifying complex VR objects. This category is the topic of this thesis and will be discussed in detail in the next chapters. The last category contains the modeling concepts that can be used for specifying behavior.

Chapter 4

Modeling Concepts for Complex Objects

In the previous chapter we have presented an overview of the VR-WISE conceptual modeling approach for Virtual Reality. So far we have seen how to specify simple objects and how to specify their position and their orientation relative to each other. However, most real-world objects are assembled of several components and often this needs to be reflected in the virtual environment. Usually, all objects connected forming a complex object, need to keep their own identity and it should be possible to manipulate them individually in the virtual environment as far as this is allowed (by their connections and constraints imposed). Therefore we need some mechanisms in our approach to specify complex objects. Complex objects can be seen as objects assembled of several simple or complex objects, called components. In this chapter we will introduce a set of high-level concepts to specify complex objects at a conceptual level in the context of the VR-WISE approach.

4.1 Introduction

We will start by giving an informal definition for a "complex object". This is done in definition 4.1.1.

Definition 4.1.1 ***(Complex objects)***
Complex objects are built from existing simple and/or complex objects. They are composed by connecting or constraining two or more simple and/or complex objects to each other. The connected objects are called components. All components keep their own identity and can be manipulated individually.

When specifying complex objects, we need some mechanisms to specify the way connections are established between the components of the complex object. A component can be a concept (instance) or a complex concept (instance). It is necessary to specify the type of connection used to connect the components because the type of connection will have an impact on the possible motion of the components with respect to each other. We explain this in more detail. Normally an object has six degrees of freedom, three translational degrees of freedom and three rotational degrees of freedom. The translational degrees of freedom are translations along the three axes of the coordinate system while the three rotational degrees of freedom are the rotations around the three axes of the coordinate system. The way components of a complex object are connected to each other may restrict the number of degrees of freedom in their displacements with respect to each other.
Note that we consider complex objects at the domain specification level as well as at the world specification level. At the domain level the abstract characteristics of a complex VR object are specified. Similar to simple VR objects, complex instances are instances of complex concepts. Also similar to the conventions we made for simple objects we will now use the following terms:

- **Complex concept:** "Complex concept" is the term that will be used for the specification of a complex VR object in the conceptual domain specification.

- **Complex instance:** The term "complex instance" will be used for instances of complex concepts which appear in the conceptual world specification.

- **Complex object:** We will use the term "complex object" when it is not necessary to distinguish between a complex concept and a complex instance.

- **Complex VR object:** Finally the term "complex VR object" will be used to refer to the visible 3D complex object in the virtual environment.

A complex concept will be graphically represented by means of a rectangle (see figure 4.1). At the top of the rectangle, a label is given to name the concept. Inside the rectangle the conceptual model for the complex concept is specified (in its graphical notation) using the modeling concepts already discussed in chapter 3 and the modeling concepts that will be introduced in this chapter.

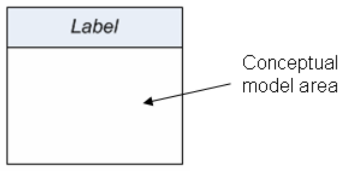


Figure 4.1: Graphical notation for the conceptual model of a Complex Concept

Complex concepts may also be used in other conceptual models, for example as a component of another complex concept. Therefore, graphically a complex concept could be referred to in the same way as a simple concept, namely by a rectangle with the label of the complex object inside it (see figure 4.2).

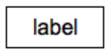


Figure 4.2: Refering to a complex concept by means of its label

A complex instance is graphically represented as an ellipse containing the instantiated conceptual specification of the complex concept it is an instance of. At the top of the ellips the name of the complex instance and the name of the complex concept it is an instance of are given using the format: "ComplexConcept : ComplexInstance". This graphical notation is illustrated in figure 4.3. We will discuss complex instances in more detail in section 4.7.

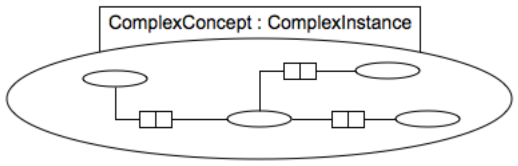


Figure 4.3: Graphical notation for an instance of a complex concept

When we look to existing VR modeling tools such as toolkits and authoring tools we see they all provide in some way mechanisms to connect objects

Table 4.1: Connection mechanisms in existing VR modeling tools

ODE	PhysX	MotionWorks
ball and socket joint	spherical joint	spherical joint
universal joint	prismatic joint	prismatic joint
hinge joint	revolute joint	revolute joint

to each other. Table 4.1 illustrates this for ODE, PhysX and MotionWorks. For example, ball and socket joints, spherical joints and universal joints all represent a connection between two objects over a center of motion. Prismatic, hinge and revolute joints all represent a connection between two objects over an axis of motion. Furthermore, prismatic joints limit the motion of the connected objects in such a way that they may only translate along the axis of motion while hinge and revolute joints limit the motion in such a way that the connected objects are only allowed to rotate around the axis of motion. From exploring different tools we have abstracted a number of general connection relations and constraints. We will introduce these relations in this chapter.

In the VR-WISE approach we provide three ways of connecting two components of a complex object, namely over a center of motion, over an axis of motion and over a surface of motion. Therefore we provide three high-level connection relations, which are introduced and explained in section 4.2:

- Connection Point Relation
- Connection Axis Relation
- Connection Surface Relation

As already indicated, these connection relations impose some constraints on the orientation and position of the components of a complex object with respect to each other. However, their motion can be constrained even more using constraints that can be specified on top of the connection relations. For example, suppose two objects are connected along a connection axis (e.g., a door connected to a door post). Then, those objects can still move along this axis or around this axis. An extra constraint could impose that they may only rotate a certain number of degrees around this axis. Therefore in section 4.3, the following connection constraints will be introduced:

- Hinge Constraint
- Slider Constraint
- Joystick Constraint

In section 4.4 we will discuss the position and orientation of a complex object. Nesting of complex objects, which means that complex objects are part of other complex objects, is discussed in section 4.5. In section 4.6 the modeling concept Role will be introduced. This modeling concept allows a concept to play different roles inside the conceptual model of a complex object. Next, in section 4.7, the instantiation of complex concepts is discussed.

There can also be constraints between objects which are not physically connected. Some examples are the simulation of a magnetic field between two objects, or a coffee cup that can be placed only on a saucer. In section 4.8 we will introduce and explain constraints that can be used in the context of what we will call *connectionless object groups*:

- Fixed Relative Position Constraint
- Fixed Relative Orientation Constraint
- Positioning Constraint

In section 4.9 we will define a number of modeling concepts which can be used to model *complex shapes*. For complex objects, all objects connected to form the complex object keep their own identity and can all be manipulated in the virtual environment as far as their connections and constraints allow. For complex shapes, all objects used to form the complex shape are melted together so they actually form one whole. So within complex shapes the connected objects loose their identity. The modeling concepts introduced in section 4.9 are based on the operators from Constructed Solid Geometry [52] (CSG will be explained later in this chapter):

- Union Relation
- Intersection Relation
- Difference Relation

4.2 Specifying connections between objects

In this section, high-level concepts for specifying connections between components of a complex object will be discussed in detail. As mentioned in the introduction, three different types of connections are distinguished: a connection over a center of motion, over an axis of motion and over a surface of motion. These types of connections are specified by means of the so-called connection relations. Next to the informal definitions of these connection relations, also a graphical notation for these relationships is given.

4.2.1 The Connection Point Relation

A first way of connecting two components of a complex object to each other is over a center of motion. In the real world we can find examples of physical objects connected over a center of motion like the shoulder of the human body connecting the arm to the torso.
A center of motion means that in the visual representation of both connected components there is somewhere a point that needs to fall on the same position during the complete lifetime of the complex VR object. We will call this point the *connection point*. Connecting two objects over a center of motion removes all three translational degrees of freedom of the objects with respect to each other.

To allow modeling (at a high level of abstraction) a connection over a center of motion between two objects, we have introduced the *Connection Point Relation*. Definition 4.2.1 gives an informal definition for the *Connection Point Relation*.

Definition 4.2.1 ***(Connection Point Relation)***
A connection point relation specifies a connection over a center of motion between two components of a complex object. This means that both components are constrained in such a way that on each of the components there is a point that needs to fall on the same position during the complete lifetime of the complex VR object of which the connected components are part.

We will now explain how this relation should be specified and how it is represented in the graphical notation of our approach.

To specify a connection point relation between two objects we have to define the connection point on each object. To define a connection point on an object we start from the position of the object. The position of the connection point is specified relative to the position of the object. The designer can do this by using the spatial relations as introduced in chapter 3. For example, the connection point lies 3 meters left of the positioning point. Directions may be combined taking into account the same rule as we have seen in chapter 3. This rule states that a combined direction combines at least two and at most three directions with the restriction that no two directions can come from the same set of directions defined as follows:

A={”front”, ”back”}
B={”left”, ”right”}
C={”bottom”, ”top”}

For example, we may translate the connection point towards the ”frontleft” direction but we may not do a translation towards the ”frontback” direction since the ”front” and ”back” directions belong to the same set. After all, it doesn’t make sense to use a direction like ”frontback” since the combined directions are exactly each others opposite.

Graphical notation

The *Connection Point Relation* is represented in our graphical notation by a rounded rectangle containing the connection point icon (see figure 4.4). The connection point relation connects two concepts (or instances). Note the use of the arrow. This arrow turns the relation into a directed relation with a source and a target concept. In figure 4.4 the source is concept A while the target is concept B. This means that A will be connected to B. Note that this is not the same as connecting B to A using the same relation. Let us explain this. Suppose both concepts already have a default position. By introducing the relation between both concepts, it is necessary to know which concept must eventually be repositioned so that the connection points of both concepts fall together. In this case it will be the concept A that will be repositioned so that its connection point falls together with the connection point for concept B. Actually, the direction of the arrow indicates the way the connection point relation must be read. Hence, figure 4.4 can be read as: *Concept A is connected by means of a connection point to concept B.*
Note that the graphical notation given in figure 4.4 does not specify the actual connection points. To be able to specify the connection points or other properties of the relation, the graphical notation is expandable. Using the expanded graphical notation the designer can specify the details of

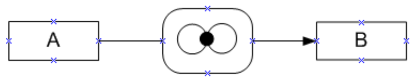


Figure 4.4: Graphical notation for the Connection Point Relation

the relation using a simple and intuitive markup language. Note that this approach will be used for all types of connection relations and all types of constraints described in this chapter. The expanded area for a connection point relation has three sub areas. The top area can hold attributes specifying general properties of the relation (such as the stiffness). The second and third areas hold the definition of the connection point for the source object respectively for the target object. The area for the connection point specification for the source object is indicated with an S inside a circle, the area for the connection point specification of the target object contains a T inside a circle. The expanded graphical notation is illustrated in figure 4.5

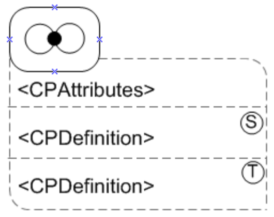


Figure 4.5: Expanded graphical notation for the Connection Point Relation

To define the attributes and the connection points, a simple markup language is used. Its syntax is given in BNF (Backus-Naur Form [1]):

```
<CPDefinition> ::= ’connection point is position point’<translation>*
<translation> ::= ’translated’ <distance> ’to’ <direction>
<direction> ::= <A>[<B>][<C>] | <B>[<C>] | <C>
<A> ::= ’front’ | ’back’
<B> ::= ’left’ | ’right’
<C> ::= ’top’ | ’bottom’
<distance> ::= <arithmetic expression>

<arithmetic expression> ::=
        <constant> | <object property> |
        ( < arithmetic expression > ) |
```

[1] http://www.garshol.priv.no/download/text/bnf.html

```
        <arithmetic expression> <operator> <arithmetic expression>

<operator> ::= + | - | * | /
<object property> ::= letter [<rest>]
<constant> ::= float-literal
<rest> ::= digit [<rest>] | letter [<rest>]
```

A connection point is specified by means of zero or more translations of the position point of the object. A translation is specified by the sentence 'translated by *distance* to *direction*' where *distance* and a *direction* need to be replaced by a distance and direction respectively. The distance is expressed by means of an arithmetic expression. Note that inside the arithmetic expression, object properties can be used. They refer to properties of the connected concepts. Referring to a property of a concept is done by means of the pattern *ConceptName.PropertyName*. Note that the use of arithmetic expressions will be the same in all connection relations. As already explained, the direction can be a combined direction. This way 'translated 2 to left' specifies a translation of the position point 2 units towards the left in order to define the connection point.
The syntax (in BNF) of the stiffness attribute is given as follows:

```
<CPAttributes> ::= [<stiffness>]

<stiffness> ::= 'connection stiffness is' <stiffnessType>
<stiffnessType> ::= 'soft' | 'medium' | 'hard'
```

Currently there is only one connection point relation attribute defined, namely the stiffness. The stiffness is specified by the sentence 'connection stiffness is' followed by the value 'soft', 'medium' or 'hard'.

Connection point relation example

Figure 4.6 gives an example a connection point relation. The handle of a joystick is connected to the base of the joystick by means of a connection point. We have defined the connection point for the target concepts, the base of the joystick, as 2 units backwards and 2 units towards the top from the positioning point of the base. The connection point for the source concept, the handle, is 4 units towards the bottom from the positioning point of the handle. The connection point relation has a stiffness property with value *soft*.

Figure 4.7 shows the connection point for the base in the example. In case the base is represented as a box and the position point corresponds to

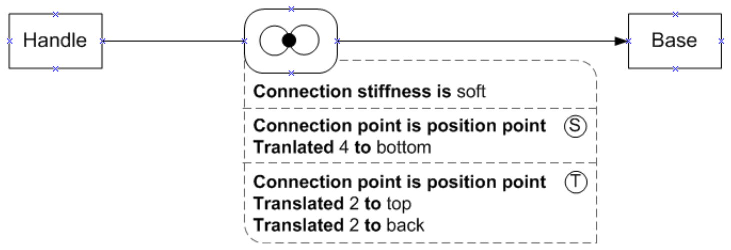

Handle
Base
Connection stiffness is soft
Connection point is position point (S)
Tranlated 4 to bottom
Connection point is position point (T)
Translated 2 to top
Translated 2 to back

Figure 4.6: connection point relation example

the middle point of this box. The connection point is indicated in red.

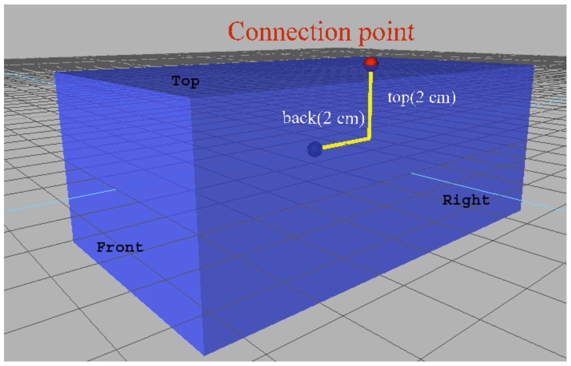


Figure 4.7: illustration of the connection point definition for the joystick base

4.2.2 The Connection Axis Relation

A second way to connect two components of a complex object is over an axis of motion. Again a lot of examples of this connection type can be found in the real world. Some examples are a wheel that turns around a certain axis; a door connected to a wall that can be opened around a certain axis; the slider of an old-fashioned typing machine that moves along a certain axis.

Actually, an axis of motion means that there is an axis that restricts the displacements of the connected objects with respect to each other in such a way that the connected objects may only move along this axis or around this axis. The axis of motion is called the *connection axis*. A connection by means of a connection axis thus removes four degrees of freedom leaving only one translational degree of freedom and one rotational degree of freedom.

In the following paragraphs we will explain the *Connection Axis Relation* introduced to model this kind of connection. First we will give an informal definition for the *Connection Axis Relation* (see definition 4.2.2).

Definition 4.2.2 ***(Connection Axis Relation)***
A connection axis relation specifies a connection over an axis of motion between two components of a complex object. This means that both components are constrained in such a way that they can only rotate around or move along their connection axis with respect to each other.

To specify a connection axis relation between two objects, we actually have to specify an axis for each of the two objects. These two axes need to fall together during the complete lifetime of the connection. Such an axis is defined as follows. Three planes through each object are defined. These are the horizontal plane, the vertical plane and the perpendicular plane. They are defined as follows:

- The *horizontal plane* is defined by the front-to-back and the left-to-right axes.
- The *vertical plane* is defined by the front-to-back and the top-to-bottom axes.
- The *perpendicular plane* is defined by the left-to-right and the top-to-bottom axes.

These planes are illustrated in figure 4.8.

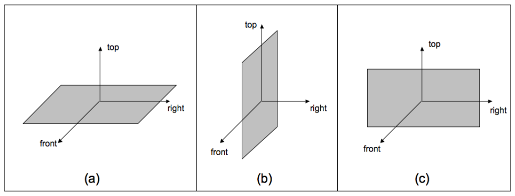


Figure 4.8: (a) the horizontal plane; (b) the vertical plane; (c) the perpendicular plane

A connection axis is defined as the intersection between two of these planes. The three predefined planes can also be translated or rotated which allows more possibilities to define an axis. Each plane may rotate over two possible axes. The horizontal plane may rotate over the left-to-right axis or the front-to-back axis; the vertical plane may rotate over the front-to-back or the top-to-bottom axis; and the perpendicular plane over the top-to-bottom or the left-to-right axis.

Next to define the connection axes, it is also necessary to give the initial positions of both components. For this we use *translation points.* A translation point of the connection axis for an object is defined as the orthogonal projection of the middle point (or position) of this object onto the defined connection axis. By default, the objects connected via a *Connection Axis Relation* will be positioned in such a way that their connection axis falls together as well as their corresponding *translation points.* However, our approach also allows the designer to translate the objects to be connected along the connection axis. This is done by means of a translation of the *translation point* along the connection axis. This principle is illustrated in figure 4.9. The red and blue cubes are connected over the yellow connection axis. The black line segments illustrate the orthogonal projections of the middle points of the cubes on the defined connection axis. Figure 4.9(a) shows the default position while figure 4.9(b) shows the positions of the connected VR objects in case the designer has specified a translation of the *translation point* of the blue cube of 1 meter backwards. Note that there may be different ways to accomplish the same result. If we would for example use a translation of the *translation point* of the red cube of 1 meter to the front, this would give the same result.

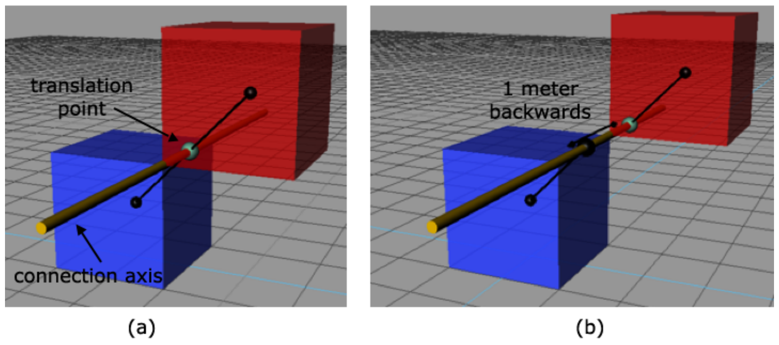


Figure 4.9: translation of the translation point on a connection axis relation

Graphical notation

The *Connection Axis Relation* is represented in our graphical notation by a rounded rectangle containing the connection axis icon. Similar to the connection point relation, the connection axis relation connects two concepts (or instances). The graphical notation for the *Connection Axis Relation* is shown in figure 4.10. Note that also here the direction of the arrow indicates the way in which the connection axis relation must be read. So in figure 4.10 we can read the relation as: *Concept A is connected by means of a connection axis to concept B.*

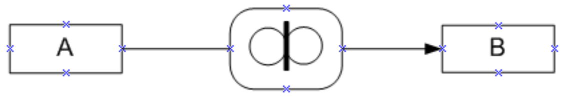


Figure 4.10: Graphical notation for the Connection Axis Relation

Again, for the specification of the connection axes or general properties of the relation, the graphical notation is expandable. When expanding the graphical notation for a connection axis relation, we get three areas similar as in case of the connection point relation. The top area is used for specifying general properties (attributes) of the relation. The second and third areas hold the definition of the connection axis for the source and the target. The area where the connection axis for the source object is specified is indicated with an S inside a circle, the area for the target object connection point specification contains a T inside a circle. The expanded graphical notation is illustrated in figure 4.11

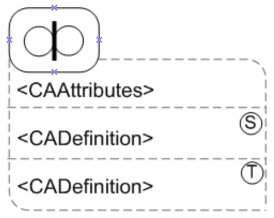


Figure 4.11: Expanded graphical notation for the Connection Axis Relation

The syntax of the language that allows specifying the attributes and the connection axes is as follows (given in BNF):

```
<CADefinition> ::= 'connection axis is intersection of:  "
                   <planeDefinition>
                   <planeDefinition>
                   [ <translationPoint> ]

<planeDefinition> ::= <horizontal> | <vertical> | <perpendicular>

<horizontal> ::=
    'horizontal plane ' [ <horizontalTrans> ] [ <horizontalRot> ]
<vertical> ::=
    'vertical plane ' [ <verticalTrans> ] [ <verticalRot> ]
<perpendicular> ::=
    'perpendicular plane ' [ <perpendTrans> ] [ <perpendRot> ]

<horizontalTrans> ::= 'translated ' <distance> 'to ' ('top' | 'bottom')
<verticalTrans> ::= 'translated ' <distance> 'to ' ('left' | 'right')
<perpendTrans> ::= 'translated ' <distance> 'to ' ('front' | 'back')


<horizontalRot> ::=
    'rotated over ' ('frontToBack' | 'leftToRight') 'axis with ' <angle>
<verticalRot> ::=
    'rotated over' ('frontToBack' | 'topToBottom') 'axis with ' <angle>
<perpendRot> ::=
    'rotated over' ('leftToRight' | 'topToBottom') 'axis with ' <angle>


<angle> ::= <arithmetic expression>

<translationPoint> ::=
      'translation point translated ' <distance> 'to ' <direction>


<CAAttributes> ::= [<stiffness>]

<stiffness> ::= 'connection stiffness is' <stiffnessType>
<stiffnessType> ::= 'soft' | 'medium' | 'hard'
```

So, a <CADefinition> specifies the connection axis as the intersection of two planes. These planes are specified by means of <planeDefinition>.
A <planeDefinition> can be either the *horizontal*, *vertical* or *perpendicular* plane followed by an optional translation and an optional rotation. As explained earlier, each plane can be translated in two directions and rotated over two possible axes. The horizontal plane can be translated to the top or bottom direction and rotated over the front-to-back or the left-to-right axes. The vertical plane can be translated to the left or right direction and rotated over the front-to-back or the top-to-bottom axis. The perpendicular plane can be translated to the front or back direction and rotated over

the left-to-right or the top-to-bottom axis. <angle> specifies an angle by means of an arithmetic expression. <distance> is defined as for the connection point relation. The connection axis definition can also have an optional <translationPoint> specification.
The syntax for the connection axis relation attributes, <CAAttributes> is the same as for the connection point relation.

Connection axis relation example

Figure 4.12 shows an example of two concepts connected to each other by means of a connection axis. A door is connected to a doorpost using a connection axis. Note the use of arithmetic expressions in the specifications of the connection axis. The connection axis on the door side, the source side, is specified by translating the vertical plane over half of the width of the Door to the right and the perpendicular plane over half of the depth of the Door to the front. The connection axis on the door-post side, the target side, is specified by translating the vertical plane over half of the width of the door-post to the left and the perpendicular plane over half of the depth of the door-post to the front. We didn't need to translate the translation point as we use the default positioning of both concepts according to the axis. The stiffness attribute of the connection axis relation is set to *medium*.

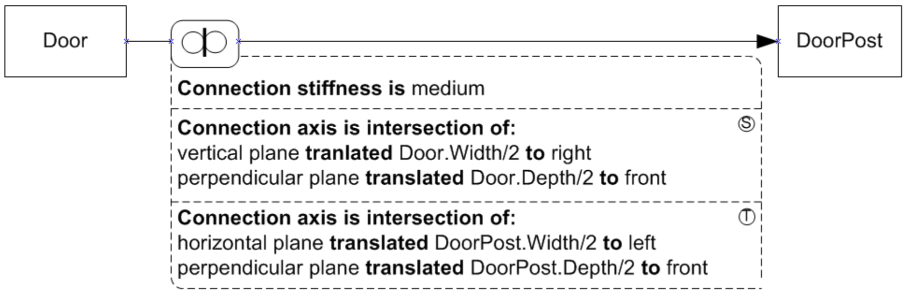

Door
DoorPost
Connection stiffness is medium
Connection axis is intersection of:
S
vertical plane **tranlated** Door.Width/2 **to** right
perpendicular plane **translated** Door.Depth/2 **to** front
Connection axis is intersection of:
T
horizontal plane **translated** DoorPost.Width/2 **to** left
perpendicular plane **translated** DoorPost.Depth/2 **to** front

Figure 4.12: connection axis relation example

Figure 4.13 illustrates the connection axis specification given in figure 4.12. Figure 4.13(a) shows the door, (b) shows the perpendicular and the vertical plane used to define the connection axis, (c) shows the translated planes and (d) shows the connection axis on the door as defined by the intersection of the translated planes.

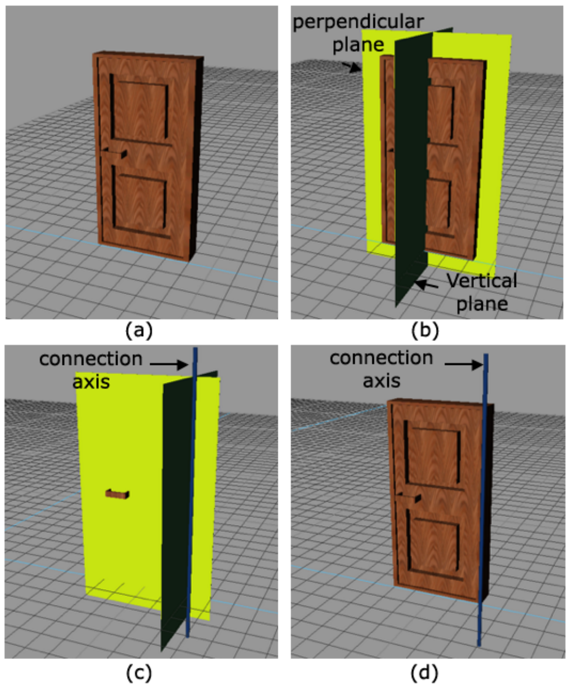


Figure 4.13: Illustration of the connection axis specification for the door object

4.2.3 The Connection Surface Relation

The third way to connect two objects to each other is over a surface of motion. A real world example of this type of connection is a boat on a water surface. The boat is able to float over the water surface. However its bottom surface stays inside the water surface. A surface of motion means that there is a surface that allows the connected objects to move along the directions of this surface. This surface removes 3 degrees of freedom. The only degrees of freedom for the objects with respect to each other are the two translational degrees of freedom in the directions of the surface and one rotational degree of freedom around the axis perpendicular to the surface. This is illustrated in figure 4.14. The surface of motion is called the connection surface. In order to be able to specify this kind of connection we have introduced the Connection Surface Relation. Definition 4.2.3 gives an informal definition of the *Connection Surface Relation*.

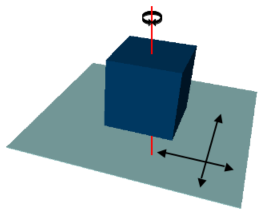


Figure 4.14: degrees of freedom for the connection surface relation

Definition 4.2.3 ***(Connection Surface Relation)***
A connection surface relation specifies a connection over a surface of motion between two components of a complex object. This means that both components are constrained in such a way that they can only move over the connection surface with respect to each other.

To specify connection surface relation we need to specify the connection surface on both objects that need to be connected. The connection surfaces on both objects need to fall together during the complete lifetime of the connection. To do this, we use the same planes as defined for the Connection Axis Relation, namely the horizontal plane, the vertical plane and the perpendicular plane. For each of the objects, the designer selects an initial plane to work with. This plane can be translated and rotated.
Similar as in the Connection Axis Relation we also need translation points to specify the initial position of both objects along the connection surface. By default, the translation point of the objects is the orthogonal projection of the middle point (position) of each object on the corresponding connection surface. The objects connected via a Connection Surface Relation will be positioned in such a way that the connection surfaces fall together as well as the translation points. Also for the connection surface relation, the translation point can be translated to specify other positions. This principle is illustrated in figure 4.15. In figure4.15(b) the translation point of the blue cube is translated 0.5 meter towards the back and 0.5 meter towards the right side.

Graphical notation

The Connection Surface Relation is represented graphically by means of a rounded rectangle containing the connection surface icon (see figure 4.16). Again, the direction of the arrow indicates the way the relation must be

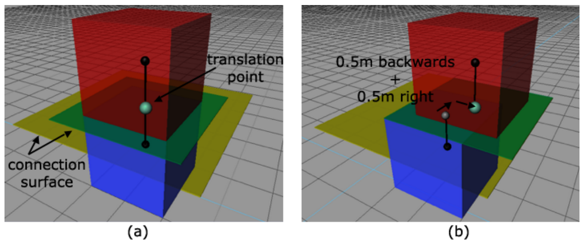


Figure 4.15: translating the translation point in a connection surface relation

read. Hence we can read the connection surface relation in figure 4.16 as follows: *Concept A is connected by means of a connection surface to concept B.*

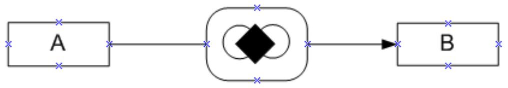


Figure 4.16: graphical notation for the connection surface relation

Similar to the connection point relation and the connection axis relation, the graphical notation of the connection surface relation is expandable. The expanded graphical notation again has three area's, one for specifying the properties of the relation, one area for the specification of the connection surface on the source object and one area for the specification of the connection surface on the target object. The extended graphical notation is illustrated in figure 4.17.

The syntax for specifying the connection surface for the source and target side is as follows (given in BNF):

```
<CSDefinition> ::= 'connection surface is:  '
                   <planeDefinition> [ <CSTranslationPoint>]


<CSTranslationPoint> ::= 'translation point translated '
                         <distance> 'to ' <direction>
                         [ 'and ' <distance> 'to ' <direction>]
```

So the specification of a connection surface consists of the sentence 'connection surface is: ' followed by a <planeDefinition>. <planeDefinition> has

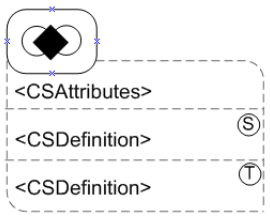


Figure 4.17: Extended graphical notation for the Connection Surface Relation

already been defined for the connection axis relation. The connection surface definition can also have a translation point specification. The specification of the translation point differs from the one of the connection axis relation because here it is possible to move the translation point in two directions on the connection surface.

Connection surface example

Figure 4.18 models the example of a boat on a water surface. The connection surface for the boat object is defined as the horizontal plane translated towards the bottom of the boat; the connection surface for the water surface is also the horizontal plane. As the water surface is a plane itself we dont need to translate the horizontal plane towards the top surface of the water. Figure 4.19 shows a possible outcome of the specification given in figure 4.18. The connection surface of the boat is allowed to move freely on the blue water surface.

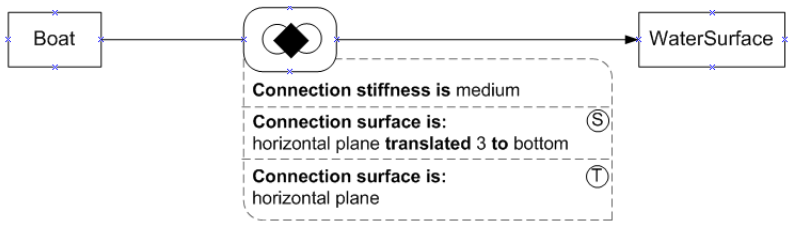


Figure 4.18: connection surface relation example

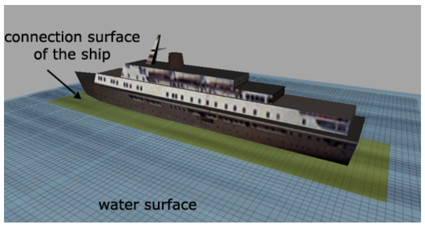


Figure 4.19: ship moving over a water surface

4.3 Constraints on connections

So far we have seen a number of relations that can be used to connect two objects to each other. As discussed, each of these relations imposes a limitation to the degrees of freedom of the connected objects with respect to each other. In this section we will discuss a number of additional constraints that allow further restricting the position and orientation of the connected objects relative to each other. Note that in the literature, the terms constraints and joints are often used interchangeably. In our research, we will systematically use the term constraint. In the following sections the *Hinge Constraint*, the *Slider Constraint* and the *Joystick Constraint* will be introduced and explained. The names of the constraints are metaphor-based which should make it easier for non-technical persons to understand and remember their meaning.

4.3.1 The Hinge Constraint

The first constraint is the *hinge constraint*. This constraint can only be placed on top of a *connection axis relation*. The *hinge constraint* still allows the objects connected through a connection axis relation to rotate around the connection axis, but forbids them to move along the axis (which would be possible without this constraint). As the name suggests, a *hinge constraint* could for example be used to attach a door to a doorpost. When specifying a *hinge constraint* the designer can also restrict the possible rotation by specifying limits for the rotation. These limits indicate the number of degrees that the connected concepts (instances) are allowed to rotate around the connection axis. The hinge limits can be specified for the clockwise direction and for the counterclockwise direction. The specified hinge limits are expressed from the initial position of both concepts (instances). When no limits are specified, the objects are allowed to rotate infinitely clockwise and counterclockwise around the connection axis. Definition 4.3.1 gives the informal definition of a hinge constraint.

Definition 4.3.1 ***(Hinge constraint)***
A hinge constraint constrains two components of a complex object connected by means of a connection axis in such a way that they are only allowed to rotate around the connection axis with respect to each other. A hinge constraint can have limits indicating how much the components may rotate around the connection axis in the clockwise as well as in the counterclockwise direction.

To have an unambiguous meaning of clockwise and counterclockwise the designer also needs to specify from which viewpoint (direction) he looks to the objects involved when using the terms clockwise and counterclockwise. If we take the reference frame of one of the connected objects then we can split the environment in eight subspaces and each of these subspaces can be characterized by means of three directions as illustrated in figure 4.20(a), e.g., (top, back, right). Now there are three different possibilities for the connection axis:

- The connection axis can go through two of the eight subspaces. This situation is shown in figure 4.20(b). This means that the designer can use as a viewpoint all possible directions (as defined earlier). Indeed, each direction is involved in the characterization of the two subspaces. E.g., if the designer would for example use *top* as his viewpoint than we could look to the axis from the (top, front, right) subspace.
- A second possibility for the connection axis is that it coincides with a plane defined by two axes of the reference frame (or a plane that is parallel with it). In this case only two pairs of directions can be used as a viewpoint, namely the directions from the axes defining the plane. In the example in figure 4.20(c) this means the designer could use front, right, back and left as a viewpoint since the connection axis is lying in the plane described by the left-right axis and the front-back axis.
- The third possibility (see figure 4.20(d)) is that the connection axis coincides with an axis of the reference frame (or is parallel with one of them). In this case only the directions of this axis can be used as a viewpoint. In the example this would mean that only left or right can be used as a viewpoint.

Once we know the viewpoint we can easily define the meaning of clockwise and counterclockwise. Because of the way in which the viewpoint is specified (as shown in figure 4.20) we know that the designer looks along the axis used for the hinge constraint. This principle is illustrated in figure 4.21.

When we look along an axis as shown in figure 4.21 we see the axis as a dot. Now we can define clockwise and counterclockwise rotations around the axes as shown in figure 19. The red spot in figure 4.22 is the axis as seen from the viewpoint.
Note also that the specified viewpoint can be different for both connected objects. Suppose the objects have a different orientation then the viewpoint

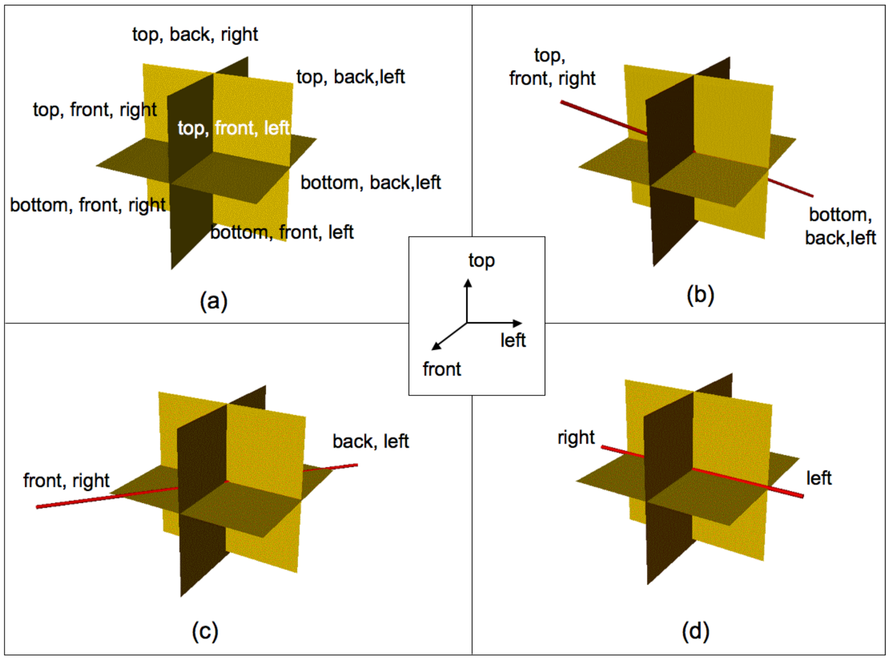


Figure 4.20: possible viewpoint on the connection axis for the specification of a hinge constraint

front could mean something completely different for the two objects. Therefore it is also necessary to specify which object's reference frame is used to determine the viewpoint. This is specified by means of the position where the hinge constraint is attached to the connection axis relation. When the hinge constraint is attached between the connection axis relation and the source object then the constraint will use the reference frame of the source object. This is similar for using the target object's reference frame.

Graphical notation

To specify this constraint the graphical notation shown in figure 4.23 is used. The constraint is represented as an octagon containing the hinge constraint icon.

Furthermore the notation is expandable (as illustrated in figure 4.24) which gives an area to specify the limits of the hinge constraint when needed.
The syntax for specifying hinge constraint limits is as follows:

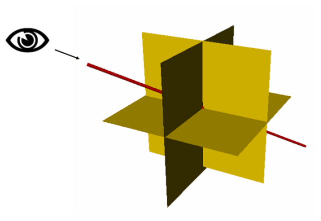


Figure 4.21: looking along an axis from the viewpoint

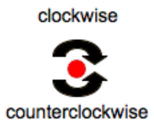


Figure 4.22: Clockwise and counterclockwise as seen from the viewpoint

```
<HingeDefinition> ::=
  'seen from ' <direction>
  'components can rotate around the connection axis:  '
  <angleLimit> 'degrees clockwise'
  <angleLimit> 'degrees counterclockwise'
```

```
<angleLimit> ::= <arithmetic expression>
```

Inside the <HingeDefinition> 'seen from <direction>' specifies the viewpoint, where <direction> is as defined earlier. Next '<angleLimit> degrees clockwise' and '<angleLimit> degrees counterclockwise' specify the limits of the hinge constraint. The limit angle is an arithmetic expression.

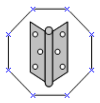


Figure 4.23: graphical notation for the hinge constraint

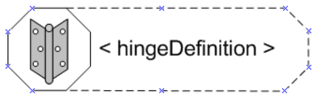


Figure 4.24: expanded graphical notation for the Hinge Constraint

Hinge constraint example

For an example of the hinge constraint we return to the door example from section 4.2.2. Figure 4.25 shows the door together with the orientation of the door and the connection axis defined on it.

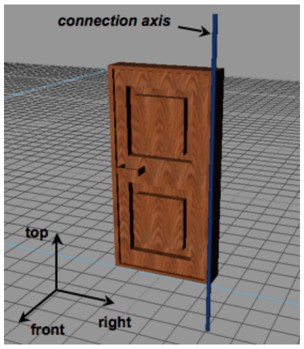


Figure 4.25: The connection axis for a door in a virtual environment

Suppose we want to specify the fact that the door can be pulled open (towards the front) 90 degrees. We can model this by specifying a maximum rotation of 90 degrees counterclockwise and 0 degrees clockwise as seen from the top. This means that from the initial position modeled (the closed door), the door could be pulled 90 degrees and cannot be pushed. By putting the constraint on the Connection Axis Relation at the side of the Door concept we indicate that the reference frame used by the constraint is the reference frame of the Door concept. The graphical representation of the conceptual model for this example is shown in figure 4.26

4.3.2 The Slider Constraint

Next to the hinge constraint there is a second type of constraint that can be specified on top of a connection axis relation. This constraint is the

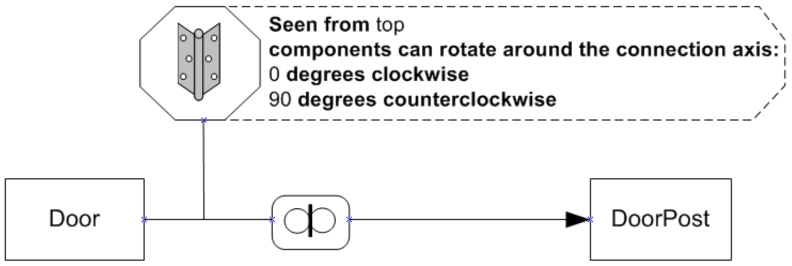

Seen from top
components can rotate around the connection axis:
0 degrees clockwise
90 degrees counterclockwise
Door
DoorPost

Figure 4.26: hinge constraint example

Slider Constraint. The slider constraint limits the movement of the objects connected by means of a connection axis in such a way that they are only allowed to move along this axis. Here, also limits can be set so that the movement is allowed only over a certain distance along the axis. When no limits are specified, the connected objects are allowed to move infinitely along the connection axis with respect to each other. An example is given by means of a robot arm that can 'slide' along a rail as illustrated in figure 4.27. In this example, the connection axis is on top of the rail and parallel with it.

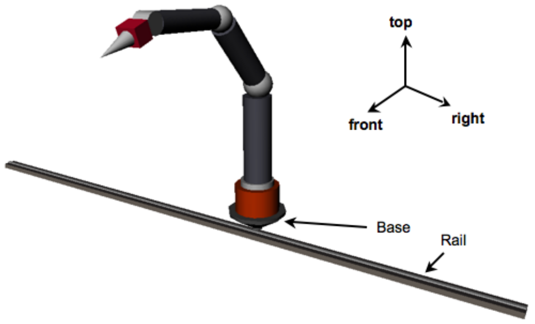


Figure 4.27: Example of a robot arm moving along a rail

First we will give an informal definition for the slider constraint. This is done in definition 4.3.2.

Definition 4.3.2 ***(Slider constraint)***
A slider constraint constrains two components of a complex object connected

by means of a connection axis in such a way that they are only allowed to move along the connection axis with respect to each other. A slider constraint can have limits indicating how much the components may move along the connection axis.

When specifying how far the objects may move along the connection axis in a certain direction, the designer has to respect the rule that the directions have to be opposite. If for example, he uses 'left' for one direction of movement, then the other direction must be 'right' and cannot e.g. be 'top'.

Graphical notation

The graphical notation for the slider constraint is given in figure 4.28. The constraint is represented as an octagon containing the slider constraint icon.

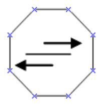


Figure 4.28: graphical notation for the slider constraint

Again, it is possible to have an expanded version of the graphical notation which allows the designer to specify the limits. This notation is illustrated in figure 4.29.

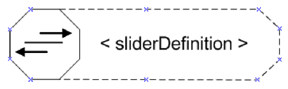


Figure 4.29: expanded graphical notation for the slider constraint

The syntax that must be used for the specification of the limits is as follows:

```
<sliderDefinition> ::=
     'components can move along the connection axis'
     (<leftRightSliding> | <frontBackSliding> | <topBottomSliding>)

<leftRightSliding> ::= <distance> 'to left'
                       <distance> 'to right'
<frontBackSliding> ::= <distance> 'to front'
                       <distance> 'to back'
<topBottomSliding> ::= <distance> 'to top'
                       <distance> 'to bottom'
```

So <sliderDefinition> exists of two limit specifications, one in each possible direction. Note that the BNF specification enforces that the directions, in which the objects can slide, are opposite. <distance> and <direction> were defined earlier.

Slider constraint example

Now we can go back to our robot example. Figure 4.30 models the Base concept connected to the Rail concept by means of a connection axis. On this relation there is a slider constraint specified. This constraint allows the Base to move 3 units towards the left along the connection axis and 5 units to the right along the axis.

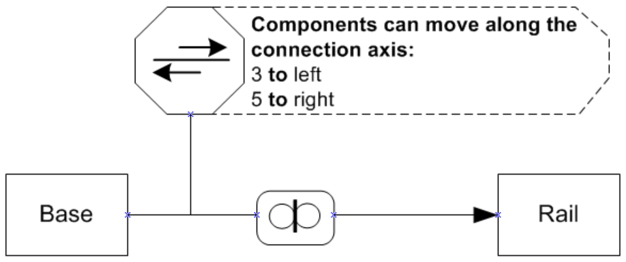


Figure 4.30: slider constraint example

4.3.3 The Joystick Constraint

The *Joystick Constraint* can be specified on top of a connection point relation. Note that this kind of constraint is also called 'universal constraint' in the domain of VR. However, we have chosen a name based on the joystick metaphor to make the constraint more intuitive for non VR-persons. The joystick constraint itself is illustrated in figure 4.31. As we can see there are two perpendicular axes specified. The connected objects are only allowed to rotate around these axes in such a way that the axes stay perpendicular to each other.
This kind of constraint is also used in a joystick (hence the name of the constraint). Figure 4.32 shows a joystick. The handle is connected to the base by means of a connection point. However in general, the handle may only turn 45 degrees towards the front, back, left and right directions.

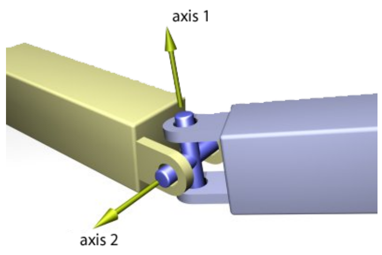


Figure 4.31: The joystick constraint [55]

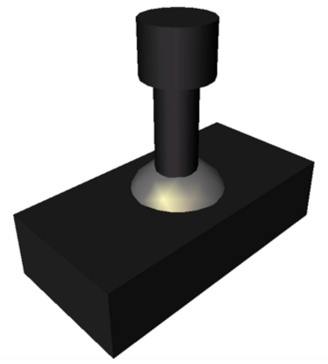


Figure 4.32: a joystick makes use of the joystick constraint

As we can see in figure 4.33(a) the handle and the joystick have a connection point. Initially we can take three axes perpendicular to each other through the connection point. These are the axes of the local reference frame of one of the connected objects placed their origin in the connection point. These are the three red axes shown in figure 4.33. These axes are called the frontToBack, leftToRight and topToBottom axis (as we have seen with the connection axis relation).

The designer can use two of these default axes through the connection point for the specification of the joystick constraint. However, to offer more possibilities, the designer can rotate these axes. Note that the axes must be rotated as one whole. This way they stay perpendicular to each other. Suppose they are rotated by 45 degrees over the leftToRight axis. This will result in the situation shown in figure 4.33(b).

Next to the specification of the two axes around which the objects may rotate, the designer can specify how much both objects may rotate clockwise and counterclockwise. Note that the limit specified for a certain axis indicates that the objects may rotate half of that limit clockwise around the axis and half of the limit counterclockwise around the axis.

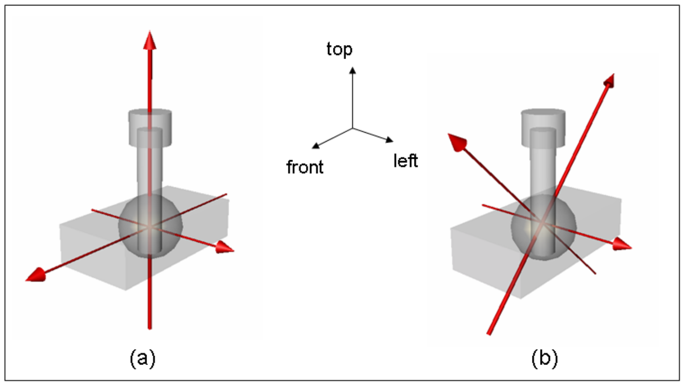


Figure 4.33: possible axes for the joystick constraint

In the joystick example, as we will see later on, the axes used will be the frontToBack and the leftToRight axes.
Before going to the graphical notation we will first give an informal definition of the joystick constraint in definition 4.3.3.

Definition 4.3.3 ***(Joystick constraint)***
The joystick constraint allows two components connected by means of a connection point to rotate with respect to each other around two perpendicular axes through the connection point. A joystick constraint can have limits indicating how much the components may rotate around the axes in the clockwise as well as in the counterclockwise direction.

Graphical notation

The graphical notation for the joystick constraint is similar to the graphical notation for the other constraints and is shown in figure 4.34. The graphical notation of the expanded version is given in figure 4.35.
The syntax of the joystick constraint specification is (in BNF):

```
<JoystickDefinition> ::=
```

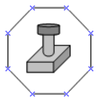


Figure 4.34: graphical notation for the joystick constraint

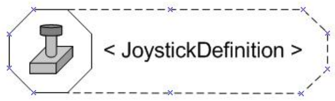


Figure 4.35: expanded graphical notation for the joystick constraint

```
    'components can rotate'
    <angle> 'degrees around' <axisDefinition>
    <angle> 'degrees around' <axisDefinition>
    [ <axesRotation>]

<axisDefinition> ::=
    'frontToBack axis' | 'leftToRight axis' | 'topToBottom axis'

<angle> ::= <arithmetic expression>

<axesRotation> ::  'axes are rotated ' <angle> 'around ' <axis>
```

In summary, the *Joystick Constraint* is specified as follows. The designer can specify the two axes used for the joystick constraint followed by the limits of the allowed rotation around these axes. Next he can also specify a rotation of the default axes.

Joystick constraint example

Now we go back to our joystick example. As we can see in figure 4.36, the Handle and the Base concepts are connected by means of a connection point relation. The joystick constraint is specified on the connection point relation at the side of the Base concept. This means that the axes used in the specification of the joystick constraint are oriented in the same way as the orientation of the Base concept's orientation. The axes used are the 'frontToBack' axis and the 'leftToRight' axis. Around both axes the objects can turn 45 degrees clockwise as well as counterclockwise. This means that the rotation limit for both perpendicular axes is 90 degrees. There is no need to have a rotation of the default axes.

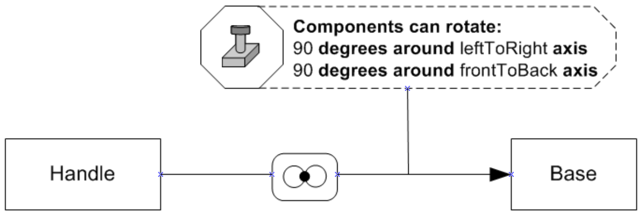

Components can rotate:
90 degrees around leftToRight axis
90 degrees around frontToBack axis
Handle
Base

Figure 4.36: joystick constraint example

Using this specification the joystick handle will be allowed to move as shown in figure 4.37.

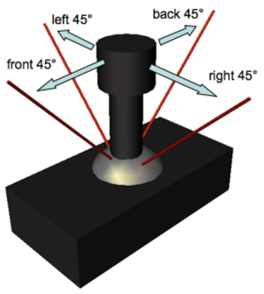


Figure 4.37: allowed movement of the handle in the joystick example

4.4 Position and Orientation of complex objects

As we have seen in the previous chapter each object has a position and orientation. This can be an explicit position and orientation or it can be relative to another object. Also for a complex object we need a mechanism to specify the position and orientation. In our approach, this is done by means of a reference object. The designer can specify that a certain component of a complex object is the reference object for that complex object. The complex object will then have the same position and orientation as its reference object. For example, if we have a complex object car with the chassis, engine and bodywork as parts, then we could specify that the chassis is the reference object. This would mean that the position for an instance of the car concept is the position of the chassis of that car-instance and that the orientation of this car instance is the same as the chassis' orientation. In the graphical representation, the reference object of a complex object is drawn in bold.

4.5 Nesting Complex Objects

In all the examples that were used so far, all components of a complex object are simple objects. However, a complex object may exist of simple and/or complex object parts. The conceptual model of a complex object can specify that another complex object is attached to another simple or complex object using the relations that were introduced in this chapter. There are several possibilities for connecting two complex objects. A first way of connecting two complex objects considers the complex objects as a whole, i.e. as if they were simple objects. In this case the connection relation (and constraints) is implied between the complex objects as such. This is illustrated in figure 4.38 where two complex concepts, *ComplexConcept1* and *ComplexConcept2*, are connected by means of a *Connection Axis Relation*. In this case the specification of the connection axis will be based on the position and orientation of the complex objects.
A second way of connecting two complex objects is by connecting one of their components. In figure 4.39 component *A* of *ComplexConcept1* is connected to component *C* of *ComplexConcept2*. Note that it is also possible to connect two complex objects by connecting one of the complex objects as a whole to a component of the other complex object.
In the previous examples, the complete graphical representation of the conceptual model of the different complex objects was used. However, this is

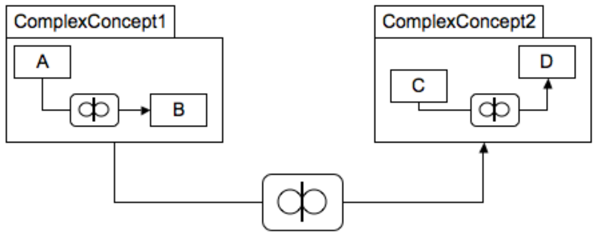


Figure 4.38: Connecting two complex objects

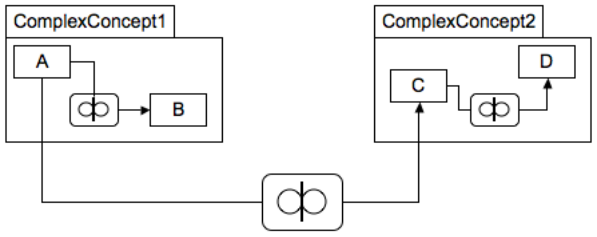


Figure 4.39: Connecting two complex objects by means of their parts

not always convenient and it may overload the models. Therefore to refer to a complex object as a whole, we can refer to in the same way as a simple object, i.e. by a box labeled with the name of the complex object. To refer to a component of a complex object, we use the box notation but labeled with *ComplexObjectName:ComponentName*. This is illustrated in figure 4.40. The example of figure 4.40 actually models the same connection as the one in figure 4.39.

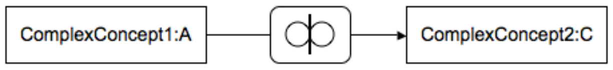


Figure 4.40: Referring to parts of complex objects for connection

This way of referring to a component of a complex object is also useful with nested complex objects. Suppose we have a complex object *Car*. One component of this complex object is the *Engine* which is a complex object on its own. The *Cylinder* is a component of the *Engine*. Now referring to the *Cylinder* component can be done using the expression *Car:Engine:Cylinder*.

4.6 Roles

As we have seen, the conceptual model for a complex concept contains a number of concepts. These concepts are the different components of the complex concept. However, it is possible that a single concept is used more than once as a component in the conceptual model of a complex concept. Take for example the complex concept Car. The concept Wheel is a part of the concept Car, but a car in general has four wheels: two front wheels, one on the left side and one on the right side, and also two rear wheels, again one on the left side and one on the right side. A possible conceptual model for the complex concept Car could be the model shown in figure 4.41. Note that the expanded notation for the connection axis relation is shown only once since the other specifications are similar.

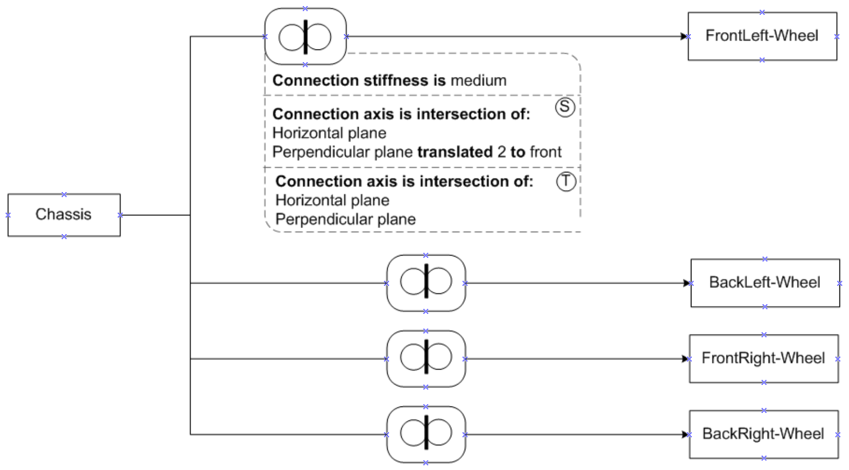

FrontLeft-Wheel
Connection stiffness is medium
Connection axis is intersection of:
S
Horizontal plane
Perpendicular plane translated 2 to front
Connection axis is intersection of:
T
Horizontal plane
Perpendicular plane
Chassis
BackLeft-Wheel
FrontRight-Wheel
BackRight-Wheel

Figure 4.41: Possible conceptual model for the concept Car

However, this model neglects the fact that the four components FrontLeft-Wheel, BackLeft-Wheel, FrontRight-Wheel, and BackRight-Wheel are actually always the same concept but each playing different roles. In this model we need to define four similar Wheel concepts, which introduce redundancy and is therefore very error-prone. Therefore, we have introduced the modeling concept *Role*. This concept is analogous to the concept role used in Object-Role Modeling (see chapter 2) where the world is described in terms of objects that play roles. Definition 4.6.1 gives the informal definition for a role in the VR-WISE approach.

Definition 4.6.1 ***(Role)***
A role is a concept playing a certain role in the context in which it is used. The concept playing the role can be defined local to the context in which the role is used or external to this context. When defined local to the context it cannot be reused in another context.

A Role is represented in our graphical notation by a double-sided rectangle. The name of the role together with the name of the concept playing the role is mentioned inside the rectangle. This graphical notation is shown in figure 4.42.

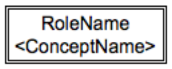


Figure 4.42: Graphical notation for the modeling concept *Role*

Using the modeling concept *Role* allows the designer to model a concept once and use it for different roles inside the context of the complex concept. For example for the concept Wheel, we introduce four roles inside the context of the complex concept Car: the role of left front wheel, right front wheel, left rear wheel and right rear wheel. The graphical notation for the conceptual model of the complex concept Car with the use of roles is given in figure 4.43.

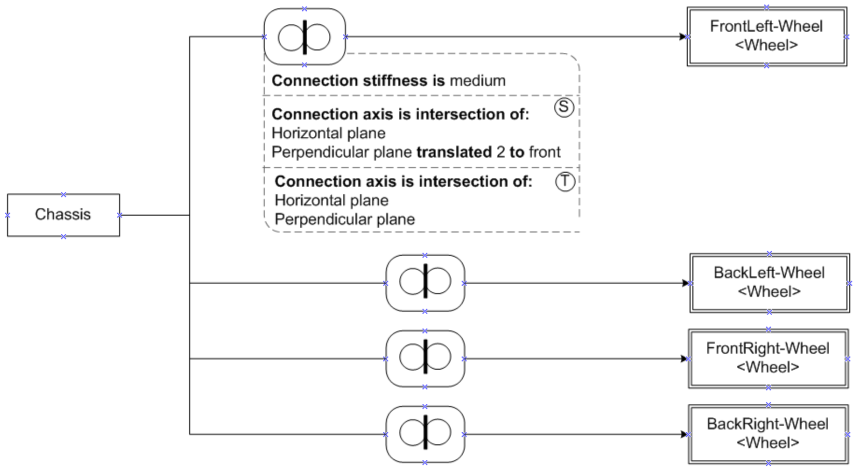

FrontLeft-Wheel <Wheel>
Connection stiffness is medium
Connection axis is intersection of: Horizontal plane Perpendicular plane translated 2 to front
S
Connection axis is intersection of: Horizontal plane Perpendicular plane
T
Chassis
BackLeft-Wheel <Wheel>
FrontRight-Wheel <Wheel>
BackRight-Wheel <Wheel>

Figure 4.43: Conceptual model for the concept Car using roles

4.7 Instantiating Complex Objects

So far we have seen how to describe complex concepts at the conceptual domain specification of the VR-WISE approach. In the conceptual world specification we need to instantiate these complex concepts in order to populate the virtual environment. The instantiation of a complex concept happens by instantiating each concept (or role) that is a component of the complex concept. In the conceptual domain specification, default values for properties of components may have been specified. These values are inherited by the instances, but in the conceptual world specification they may be overwritten. In figure 4.44 the instance SmallCar is specified as an instance of the Car concept defined in figure 4.43. Note that all concepts (and roles), which are part of the Car concept specification are now instantiated and some default values have been overwritten. For example,the default positioning of the wheels (2 meters to the front and back side of the car chassis) are overwritten and they are positioned 1.5 meters to the front and back side of the chassis. This is reflected in the Connection Axis Relation where the plane transformation is changed from 2 to 1.5 (between the SmallChassis and FLWheel instances).

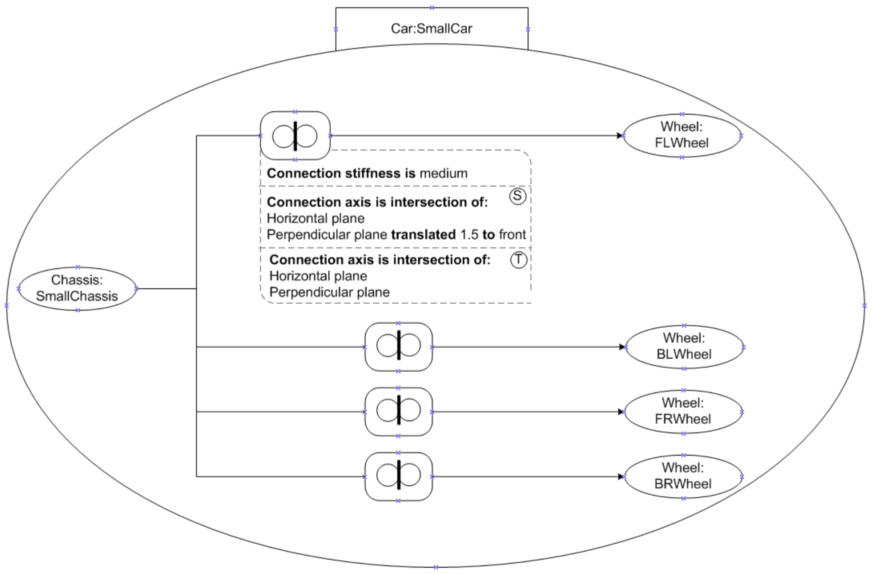

Car:SmallCar
Wheel: FLWheel
Connection stiffness is medium
Connection axis is intersection of:
S
Horizontal plane
Perpendicular plane translated 1.5 to front
Connection axis is intersection of:
T
Horizontal plane
Perpendicular plane
Chassis: SmallChassis
Wheel: BLWheel
Wheel: FRWheel
Wheel: BRWheel

Figure 4.44: SmallCar instantiated from the Car concept

4.8 Specifying and constraining connectionless groups of objects

In the previous sections we have seen how two objects can be physically connected and how their translational and rotational degrees of freedom with respect to each other can be constrained. However it may also be necessary to constrain the relative position and orientation of objects while they are not physically connected. Some examples can be found in reality, e.g., when simulating a magnetic field between two objects, or a coffee cup which can be placed on a saucer. In this section we will describe a number of constraints that can be used in the context of connectionless groups of objects.

4.8.1 The Fixed Relative Position Constraint

The *Fixed Relative Position Constraint* forces the position of one object to be constant relative to another object's position. This constraint is specified on top of a *spatial relation* as it is constraining the spatial position of the two objects given by means of the relation. However, it is also possible that this constraint holds between two objects for which no spatial relation has been specified. Definition 4.8.1 gives the informal definition for the *Fixed Relative Position Constraint.*

Definition 4.8.1 ***(Fixed Relative Position Constraint)***
The fixed relative position constraint forces two objects to keep the same position relative to each other during their complete life-time.

We will start explaining the case where a spatial relation has been defined between the objects involved. The *Fixed Relative Position Constraint* indicates that this relation should hold for the complete lifetime of both objects. In other words their relative position may not be changed. No other information needs to be provided. Therefore we have chosen to attach the constraint symbol, an octagon, containing an F (indicating 'fixed') to the graphical notation of the spatial relation between the objects. We have chosen for a constraint symbol instead of a new type of spatial relation in order to keep the number of modeling primitives low and not to clutter the diagrams. The graphical notation of the *Fixed Relative Position Constraint* is shown in figure 4.45.

The second case is when the designer wants to specify that two objects that are not related by means of an explicit spatial relation must have a fixed

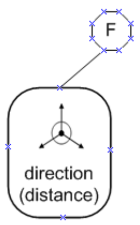


Figure 4.45: graphical notation of the fixed relative position constraint

relative position with respect to each other. We illustrate the purpose of this constraint with an example. Suppose we have two objects, A and B that are not related via a spatial relation. They both have a concrete position in the environment. However the designer wants to specify that their initial position (at time zero in the lifetime of the objects) with respect to each other must be frozen. This means that when object A undergoes a translation in the virtual environment, object B needs to undergo the same translation and vice versa. The graphical notation used in figure 4.45, is not usable here since there is no explicit spatial relation. Therefore we need to introduce an extra graphical notation for this case. This graphical notation is illustrated in figure 4.46. It is the constraint symbol (an octagon) containing the spatial relation icon. This graphical notation is placed between the two concepts that the designer wants to constraint with a fixed relative position.

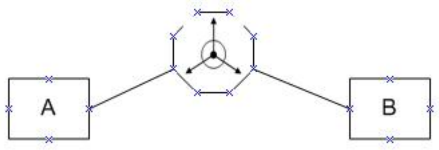


Figure 4.46: graphical notation of the fixed relative position constraint without explicit spatial relation

4.8.2 The Fixed Relative Orientation Constraint

The *Fixed Relative Orientation Constraint* freezes one object's orientation relative to another object's orientation. It is therefore modeled on top of an *orientation relation* between two objects, but it can also be used between two objects that are not related by an explicit relative orientation relation.

Definition 4.8.2 gives an informal definition for the fixed relative orientation constraint.

Definition 4.8.2 ***(Fixed Relative Orientation Constraint)***
The fixed relative orientation constraint forces two objects to keep the same orientation relative to each other during their complete life-time.

We start with the first case in which an orientation relation is specified between both objects. Again there is no need to specify additional information as the constraint only indicates that the relative orientation between two objects needs to stay constant during the complete lifetime of the objects. Therefore the fixed relative orientation constraint is represented in the same way as the fixed relative position constraint. A octagon containing an F is attached to the relative orientation between the constrained objects. The graphical notation for the fixed relative orientation constraint is illustrated in figure 4.47.

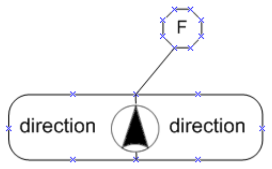


Figure 4.47: graphical notation of the fixed relative orientation constraint

Similar to what we have seen with the fixed relative position constraint, the designer may want to specify that two objects which are not related via an explicit orientation relation have a fixed relative orientation. Take two objects, A and B that are not related via an orientation relation. They both have their own orientation in the virtual environment. The designer may wish to specify that when object A undergoes a rotation, object B needs to undergo the same rotation and vice versa. To represent this constraint, we introduced a graphical notation similar to the one for a fixed relative position constraint. It consists of the constraint symbol containing the orientation relation icon and is placed between the concepts to be constrained. This graphical notation is illustrated in figure 4.48.

4.8.3 The Positioning Constraint

In a virtual environment the user can manipulate the objects, he can move and turn them. However, most VR objects cannot be positioned on an

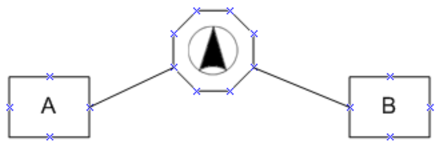


Figure 4.48: graphical notation of the fixed relative orientation constraint without explicit orientation relation

arbitrary place. This can have several reasons: physics properties, human conventions, etc. One example is a coffee cup that is mostly placed on a saucer while it can usually not be placed on a wire.
In order to describe this kind of constraint we have introduced the *Positioning Constraint*. This constraint allows describing where VR objects can be positioned and which VR objects can serve as a positioning base for other objects. This approach is based on the approach described by [54]. Definition 4.8.3 gives the informal definition for the positioning constraint.

Definition 4.8.3 ***(Positioning Constraint)***
The positioning constraint specifies which objects can serve as a positioning base for which other objects. It is specified by means of anchor area constraints and binding area constraints, which are labeled by one or more labels. Objects with an anchor area can be placed on objects with a binding area when they both share the same label.

- *An anchor area specifies for an object the area of the object that can be used for positioning.*
- *A binding area specifies for an object the area of the object that can be used as positioning base.*

We will explain the *Positioning Constraint* by means of an example. Suppose we have two concepts in the virtual environment, let's say Desk and Book. We want to express that all instances of the concept Book can be positioned only with their back side on top of each possible instance of Desk. Therefore we can define the top surface of Desk as being a *binding area*. Next, we define the backside of Book to be an *anchor area*. Anchor area's can be positioned on binding area's. However, we also need to indicate on which binding areas the anchor area of a book can be positioned. For this purpose we will label the areas. We can for example label the anchor area of the book with the label Workspace. All binding area where the book can be places are given the same label. Thus the binding area of the desk will also be labeled Workspace. Suppose books can also be placed on shelves. Then the designer could also define a binding area on the shelve concept and label it as a Workspace. Note that an area may have more than one label. For example if we would model a computer in such a way that it can only be placed on a desk. Therefore we would specify an anchor area for the computer concept. We could label it Desktop. Then the binding area of the desk can be given a second label Desktop next to the Workspace label. This way, computers can be placed on desks while books can be placed on desks and shelves.

Graphical notation

Now we will describe the graphical notation for this constraint. Again we have an expandable graphical notation. Figure 4.49 illustrates the graphical icons used for (a) the binding area and (b) the anchor area.

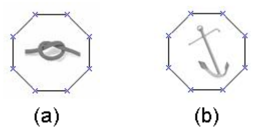


Figure 4.49: graphical icons for (a) the binding area; (b) the anchor area

In the expanded graphical notation there is an area for specifying the binding area or the anchor area. This is shown in figure 4.50.

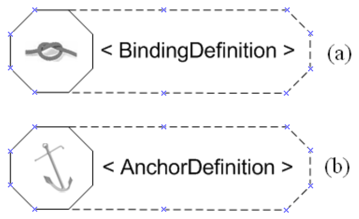


Figure 4.50: Expanded notation for (a) binding area and (b) anchor area

The syntax for the <BindingDefinition> and the <AnchorDefinition> is as follows:

```
<BindingDefinition> ::=
        'the binding surface is ' <surface> 'labeled as' <labelName>
<AnchorDefinition> ::=
       'the anchor surface is ' <surface> 'labeled as' <labelName>


<surface> ::= 'front' | 'back' | 'left' |
              'right' | 'top' | 'bottom'
<labelName> ::= <label> | '{' <labelEnum> '}'
<label> ::= <char> | <label><char>
<labelEnum> ::= <label> | <labelEnum><label>
<char> ::= 'a' | 'b' | 'c' | ...| 'z' | '0' | '1' | ...| '9'
```

Positioning constraint example

We now come back to the example of the positioning constraint between the book concept and the desk concept. Figure 4.51 shows the graphical notation for the definition of the binding area on the desk while figure 4.52 shows the graphical notation for the definition of the anchor area on the book.

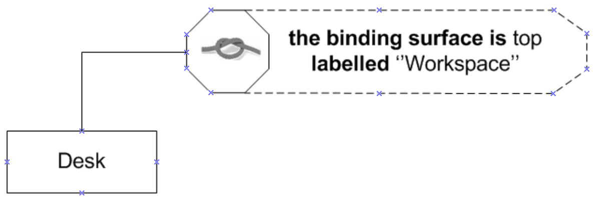


Figure 4.51: specification of the binding area on the Desk concept

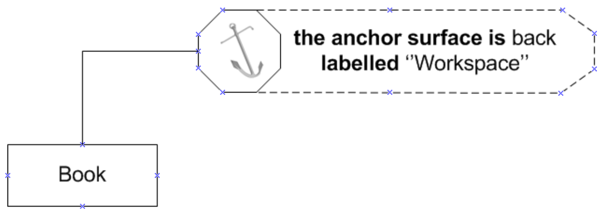


Figure 4.52: specification of the anchor area on the Book concept

4.9 Specifying complex shapes

In this section we will define a number of modeling concepts which can be used to model *complex shapes*. In the case of a complex object, all objects connected to form the complex object keep their own identity and can all be manipulated individually in the virtual environment as far as their connections and constraints allow. In case of a complex shape, all objects are melted together to form one whole. So within complex shapes the connected objects loose their identity. Only the complex shape can be manipulated in the virtual environment as one entity. Actually, when specifying complex shapes we are concerned with the representation of the complex shape in the virtual world. When specifying complex objects we are concerned with the physical structure of the complex object.
The approach we chose to model complex shapes is very closely related to Constructed Solid Geometry (CSG). Constructive solid geometry [52] is a technique used for solid modeling. This technique can be used to create complex shapes or objects using a set of boolean operators. Although other boolean operators may be offered, the boolean operators mostly used in CSG are union, intersection and difference. Figure 4.53 shows two initial VR objects. When performing the logical operators on these objects the results are as shown in figure 4.54. Figure 4.54(a) shows the union operation. Note that there is no visual difference with the original figure. However, in the original figure the two VR objects are two different VR objects while after performing the union operation they define one VR-object. Figure 4.54(b) shows the intersection of both VR objects while figure 4.54(c) shows the difference between the cube and the cylinder. Note that after performing a union, intersection or difference relation between two objects, the original objects used in the relation do not exist anymore. However, a new object is created as being respectively the union, intersection or difference of both original objects.

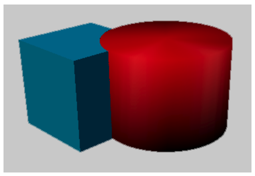


Figure 4.53: Two initial objects used in CSG operations

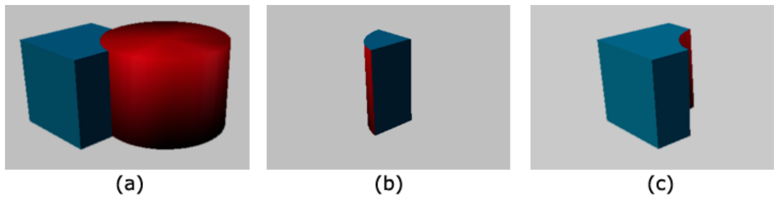


Figure 4.54: CSG operations: (a) union; (b) intersection and (c) difference

The CSG technique is popular because it allows creating very complicated geometries with relatively simple objects and operators. Therefore we have introduced three relations in the VR-WISE approach which can be used in the same way as the logical operations in CSG. In the following paragraphs we will introduce the *Intersection Relation*, the *Difference Relation* and the *Union Relation*.

4.9.1 The Union Relation

The *Union Relation* allows the designer to model a new object by taking the union of two existing objects. Definition 4.9.1 gives an informal definition for the union relation.

Definition 4.9.1 ***(Union relation)***
The union relation defines a new object as the union of two other objects, i.e. the geometry for the union of the two objects will consist of all points that are part of the geometry representing the first object and all points that are part of the geometry representing the second object.

The graphical notation for the Union Relation is shown in figure 4.55.

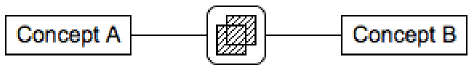


Figure 4.55: Graphical notation of the Union Relation

4.9.2 The Intersection Relation

The *Intersection Relation* can be used to model a new object by intersecting two existing objects. Definition 4.9.2 gives an informal definition of the intersection relation.

Definition 4.9.2 ***(Intersection relation)***
The intersection relation defines a new object as the intersection of two other objects, i.e. the geometry for the intersection of the two objects will consist of all points that are part of the geometry for the first object as well as of the geometry for the second object.

The graphical notation for the Intersection Relation is shown in figure 4.56.

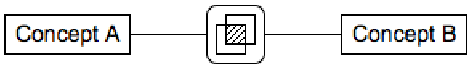


Figure 4.56: Graphical notation of the Intersection Relation

4.9.3 The Difference Relation

The *Difference Relation* allows the designer to model a new object by taking the difference between two existing objects. Definition 4.9.3 gives an informal definition for the difference relation.

Definition 4.9.3 ***(Difference relation)***
The difference relation defines a new object as the difference between two other objects, i.e. the geometry for the difference of the two objects will be represented by a geometry consisting of all points that are part of the geometry for the first object but are not part of the geometry for the second object.

Note that the *Difference Relation* is not symmetric like the *Intersection Relation* and the *Union Relation*. This means that the difference of concept A with concept B is not the same as the difference of concept B with concept A. Therefore in the graphical notation, the arc connecting the concepts through the *Difference Relation* is directed. The direction of the arc specifies the way the relation must be read. The graphical representation is shown in figure 4.57. Because of the direction of the arrow we read the relation in figure 4.57 as *"Concept A minus Concept B"* or *"The difference of concept A with concept B"*.

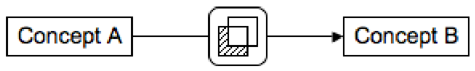


Figure 4.57: Graphical notation of the Difference Relation

4.9.4 Modeling CSG Trees

A model for a complex shape may consist of several concepts all combined using the logical operations discussed so far. Therefore in CSG a *CSG tree* is often used to give a clear overview of the model constructed. A CSG tree only shows the history of the generation of the complex shape specified. It does not show additional information like for example relative positioning of parts. The root of a CSG tree indicates the actual complex shape. An example of a CSG tree is shown in figure 4.58.

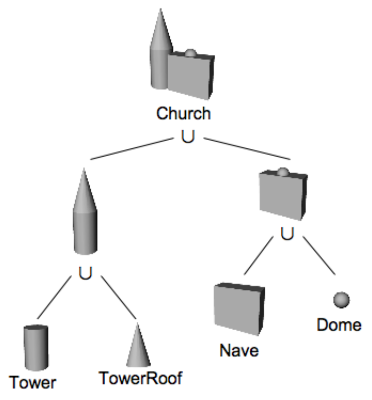


Figure 4.58: An example of a CSG tree

Note that a CSG tree is always unambiguous. However it is not unique. In other words, a CSG tree always models exactly one object but a single object can in general be modeled with several CSG trees.

The organizational structure of a CSG tree can also be reflected in the VR-WISE approach. This is done by means of nesting complex shapes. The example shown in figure 4.58 is modeled in VR-WISE as shown in figure 4.59. Note that the model represented in figure 4.59 not only shows the history of how the complex shape is created but also shows additional information like positioning of the parts of which the complex shape has been constructed.

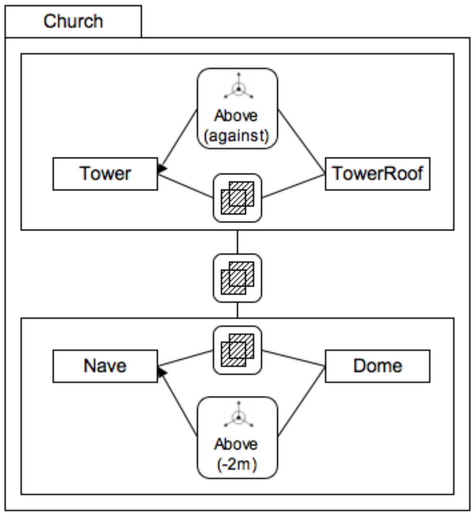


Figure 4.59: CSG tree as represented in the VR-WISE approach

As we can see in figure 4.59 the complete concept is identified with the label *Church*. Subparts don't necessarily need to have an identifying label. Only the root needs to have an identifying name.

4.9.5 Orientation and position of subparts in CSG

When two objects are only combined with the *Union*, *Intersection* or *Difference Relation* and no other information like relative positioning is provided then before the calculation of the visual representation of the complex shape in the virtual environment, both subparts are given the same orientation and the same position. This means that the position of the subparts lie on the same coordinate. This default situation can be overwritten by modeling extra information using spatial relations and/or orientation relations.

Part II

Formal Definitions

In the first part of the dissertation we have discussed a set of high-level modeling concepts that can be used for specifying VR objects (simple as well as complex) on a higher level of abstraction. For all modeling concepts we have given an informal definition. In this part of the dissertation a formal specification of these high-level modeling concepts will be given. Such a formal specification has several benefits:

- A formalization unambiguously specifies the modeling concepts and thus allows to build conceptual specifications that are also unambiguous.
- In addition, a formal specification gives a formal foundation that allows to reason about conceptual specifications.
- The formal specification for the modeling concepts is independent from any implementation. Therefore, different implementations can be built based on the formal specification.

We have opted for a logic-based formalism. More in particular, the F-logic (see chapter 5) formalism extended with arithmetic operators such as cosinus, sinus, ... will be used. These extensions are similar to the ones made by the OntoBroker tool [23]. We also include operators similar to the ones used in Flora-2 [68], e.g., we use the operator `'is'`, which is used to evaluate arithmetic operators.

This part is organized as follows. First, in chapter 5 we give a brief introduction to F-logic and we clarify why F-logic is used for the formalization of the modeling concepts. In chapter 6 a number of general classes and deductive rules are defined. These classes and rules will be used in the actual formalization of the modeling concepts. In chapter 7 all modeling concepts for modeling static virtual environments are formalized. Chapter 8 gives the formalization of the connection relations defined inside the VR-WISE approach. Next, we give the formalization of constraints in chapter 9 and CSG relations in chapter 10 respectively. Finally we give a number of possible applications of the formalization in chapter 11.

Chapter 5

Introduction to F-Logic

Frame-Logic is a frame-based language. The central modeling primitives in frame-based languages are classes with certain properties (attributes). These attributes can be used to store primitive values or to relate classes to other classes. In general classes can be sub-classed. Frame-Logic (F-Logic) is a full-fledged logic. It provides a logical foundation for object-oriented languages for data and knowledge representation. F-logic stands in the same relationship to the object-oriented paradigm as classical predicate calculus stands to relational programming. In this thesis, F-logic is used to give a formal specification of the modeling concepts developed for modeling objects, simple as well as complex. In this chapter we will give a brief introduction to F-Logic. The motivation for using F-Logic can be found in section 5.5. The remainder of this chapter is structured as follows. In section 5.1 we introduce F-Logic by means of examples. Next in section 5.2 we give an overview of the F-Logic syntax. This introduction is based on [36], [40] and [23] to which we refer the interested reader for more details. In section 5.4 we give a brief introduction to the F-logic semantics.

5.1 F-Logic by example

5.1.1 Signatures

The following F-logic statements provide information about classes and their signatures. A signature of a class specifies names of properties and the methods that are applicable to that class.

```
person[ name ⇒ string;                                    (1)
        address ⇒ string]
```

```
professor[publications ⇒⇒ article;                        (2)
          dep ⇒ department;
          highestDegree ⇒ string;
          highestDegree •→ "phd"]

article[authors ⇒⇒ person;                                (3)
        title ⇒ string;
        writtenAt ⇒ institution]

student[studentNbr ⇒ integer]                             (4)

employee[employeeNbr ⇒ integer]                           (5)

department[ assistants ⇒⇒ (student, employee);            (6)
            profs ⇒ professor;
            head ⇒ professor]
```

Properties and types

Statement (1) gives the class signature for the class *person*. It says that for every member-object of the class *person* the properties *name* and *address* must be of type string. To specify an attribute definition ⇒ is used.

Multi-valued properties

Statement (2) gives the class signature for the class *professor*. `publications ⇒⇒ article` states that publications is a multi-valued property. This means that each member-object in the class *professor* has an attribute *publications* which has a set of objects as its value. The objects in this set-value are restricted to the type *article*.

Inheritable properties

Also in (2), `highestDegree •→ "phd"` states an inheritable property for the class *professor*. Asserting an inheritable property for a class has the effect that each member-object of that class inherits this property. Suppose we have a member "*bill*" of class *professor*, then *bill* will have the property *highestDegree* with value "phd" by inheritance. An inheritable property may also be inherited by a subclass. In the case of a subclass, the inheritable property remains inheritable in this subclass while an inheritable property inherited by a member of the class becomes not inheritable.

We will discuss class membership and the subclass relation in more detail in sections 5.1.2 and 5.1.4. Note that different kinds of information about objects can be mixed in one statement. For example, `highestDegree ⇒ string` indicates that the type of the highestDegree property must be string while `highestDegree •→ "phd"` states an inheritable property for the class *Professor*.

Conjunction of types

In statement (6) we have `assistants ⇒⇒ (student, employee).` This statement specifies a multi-valued property. However, the objects that belong to the value set must simultaneous be a member of the *student* class and the *employee* class. In natural language we could say that assistants of a department must be students and simultaneously employees.

5.1.2 Class membership

In F-Logic we use ":" to represent class membership. Statement (7) denotes that *mary* is a member of the class *employee* while (8) states that *dinf* is a member of *department*.

```
mary :  employee                                          (7)
dinf :  department                                        (8)
```

(9) states that the object denoted by the oid *mary* has employeeNbr 25186.

```
mary[employeeNbr → 25186]                                 (9)
```

Note that in F-Logic classes are reified which means that they belong to the same domain as individual objects. This makes it possible to manipulate classes and member-objects in the same language. This way a class can be a member of another class. This gives us a great deal of uniformity.

5.1.3 Method signatures and deductive rules

Next to properties, classes can have methods. In the following class signature we give the signature of a method attached to the class.

```
professor[publications ⇒⇒ article;                       (10)
          dep ⇒ department;
          highestDegree ⇒ string;
          highestDegree •→ phd;
          boss ⇒ professor]
```

In (10) we give the signature of a method *boss*. Actually this signature is the signature of a new property *boss* since a property is simply a method without arguments. Indeed, in F-Logic, there is no essential difference between methods and properties. So the method *boss* takes no arguments as input and gives an object of type *professor* as output. Next, statement (11) is a deductive rule defining the new method *boss* for objects of the class *professor*. Rules offer the possibility to derive new information. They encode information of the form: when the precondition is satisfied, the conclusion also is satisfied. The precondition is called the *rulebody* while the conclusion is called the *rulehead*. Thus the part of the rule which comes before the ← sign is the head, the part of the rule after the ← sign is called the rule body. So (11) states that when a member B of type *professor* is the head of a departement D for which a member P of type *professor* is working, then B is the boss of P.

```
P[boss → B] ← P : professor ∧                         (11)
              D : departement ∧
              P[dep → D[head → B : professor]]
```

It is also possible to create methods that take some arguments as input. Syntactically the arguments are included in parentheses and are separated from the method name by the @-sign. However, when the method takes only one argument the parentheses may be omitted. Statement (12) gives the signature of a method *papers* for the class *professor*. It takes one argument of type *institution* and returns a set-value of type *article*.

```
professor[papers@institution ⇒⇒ article]              (12)
```

Next, the deductive rule in (13) defines the new method *papers* for the member-objects of class *professor*.

```
P[papers@I →→ Z] ← P : professor ∧                    (13)
                   I : institution ∧
                   P[publications →→ Z] ∧
                   Z[writtenAt → I]
```

(13) states that the method *papers* returns a set-value of type *article* containing all member-objects of the class *article* which were written by a

member-object P of class *professor* at a member-object I of class *institution*. In natural language this means that given an institution I and a professor P, the method *papers* returns all articles that were written by professor P at institution I.

5.1.4 Subclass relationship

In F-Logic we use "::" to represent the subclass relationship. Statement (14) denotes that *student* is a subclass of *person*, statement (15) says that *employee* is a subclass of *person*, while statement (16) denotes that *professor* is a subclass of *employee*.

```
student :: person                                    (14)
employee :: person                                   (15)
professor :: employee                                (16)
```

5.1.5 Function symbols

In F-Logic, function symbols can be used as constructors for class id's and class member id's. Statement (17) introduces the unary function symbol *phdStudent*. It states that a phd student of a professor is a subclass of the class student. Note that *phdStudent*(*professor*) is the logical id of a new class.

```
phdStudent(professor) :: student                     (17)
```

In (18) we introduce a new property for the class *professor*, namely the property *phdStudents*. Since *phdStudent*(*professor*), defined in (17), is the logical id of a class, we can use it as the type for the *phdStudents* property.

```
professor[phdStudents =>> phdStudent(professor)]    (18)
```

Since *phdStudent*(*professor*) is the constructor of a class id, we can construct the class *phdStudent*(*bill*). Next we can state that *mary* is a member of the newly created class *phdStudent*(*bill*). This is expressed in (19).

```
mary : phdStudent(bill)                              (19)
```

5.1.6 Predicates

In F-Logic, predicate symbols can be used in the same way as in predicate logic. They are represented by a predicate symbol followed by one or more id-term separated by commas and included in parentheses. Statement (20) gives an example of a predicate in F-Logic.

```
promotorOf(mary, bill)                                  (20)
```

An n-ary predicate symbol can usually be translated into the more natural style of modeling in F-Logic. (21) shows the encoding of the predicate illustrated in (20).

```
mary[promotor → bill]                                   (21)
```

5.1.7 Queries

Queries can be considered as a special kind of rules. They can be seen as rules with an empty head. The following query (22) requests for all members of the class *professor* which are working at the department indicated by the oid *dinf*.

```
?- X : professor ∧ X[dep → dinf]                        (22)
```

So far we have introduced the main features of F-Logic by means of some examples. In the next section we will take a look to the F-Logic syntax in more detail.

5.2 Syntax

In this section we will look in more detail to the F-Logic syntax.

5.2.1 The alphabet of an F-Logic language

Definition 5.2.1 gives the definition of the alphabet of an F-Logic lanuage $\mathcal{L}$.

Definition 5.2.1 *(Alphabet of an F-Logic language)*

The alphabet of an F-Logic language, $\mathcal{L}$, consists of:

- *a set of object constructors, $\mathcal{F}$;*
- *an infinite set of variables, $\mathcal{V}$;*
- *auxiliary symbols, such as, (,), [,], $\leftarrow$, $\twoheadrightarrow$, $\bullet\!\!\rightarrow$, $\bullet\!\!\twoheadrightarrow$, $\Rightarrow$, $\Rightarrow\!\!\!\!\Rightarrow$,* etc; and
- *usual logical connectives and quantifiers, $\vee$, $\wedge$, $\neg$, $\rightarrow$, $\forall$, $\exists$.*

The elements of $\mathcal{F}$, the object constructors, play the role of function symbols of F-Logic (see section 5.1.5). Each function symbol has an arity which determines the number of arguments a function can take. Symbols of arity 0 are called *constants* while symbols with an arity of at least 1 are used to construct larger terms out of simpler ones. An *id-term* is a first-order term composed of function symbols and variables as in predicate calculus [64].

5.2.2 Molecular formulas

A language of F-Logic consists of a set of formulae which are constructed out of symbols from the alphabet. The simplest type of formulas are $molecularF-formulas$ or simply $F-molecules$. Definition 5.2.2 gives the definition of molecular formulas.

Definition 5.2.2 *(Molecular Formulas)*

A molecule in F-logic is one of the following statements:

1. *An is-a assertion of the form $C :: D$ or of the form $O : C$; where C, D and O are id-terms.*

2. *An object molecule of the form O[";"-separated list of method expressions] where O is an id-term.*
   A method expression can be either a ***non-inheritable data expression****, an* ***inheritable data expression*** *or a* ***signature expression****.*

   - *A* ***non-inheritable data expression*** *takes one of the following two forms:*

     (a) A non-inheritable scalar expression ($k \geq 0$):
     ScalarMethod@$\mathtt{Q}_1$, ..., $\mathtt{Q}_k \rightarrow$ T

     (b) A non-inheritable set-valued expression ($l, m \geq 0$):
     SetMethod@$\mathtt{R}_1$, ..., $\mathtt{R}_l \twoheadrightarrow \{\mathtt{S}_1$, ..., $\mathtt{S}_m\}$

     where $Q_1, \ldots, Q_k$ and $R_1, \ldots, R_l$ and $S_1, \ldots, S_m$ are id-terms.

   - *An* ***inheritable data expression****, scalar or set-valued, is similar to non-inheritable data expressions. The only difference is that we replace "$\rightarrow$" by "$\bullet\!\!\rightarrow$ " and "$\twoheadrightarrow$ " by "$\bullet\!\!\twoheadrightarrow$ ".*

   - *A* ***signature expression*** *takes one of the following two forms:*

     (c) A scalar signature expression ($n, r \geq 0$):
     ScalarMethod@$\mathtt{V}_1$, ..., $\mathtt{V}_n \Rightarrow$ ($\mathtt{A}_1$, ..., $\mathtt{A}_r$)

     (d) A set-valued signature expression ($s, t \geq 0$):
     SetMethod@$\mathtt{W}_1$, ..., $\mathtt{W}_s \Rrightarrow$ ($\mathtt{B}_1$, ..., $\mathtt{B}_t$)

     where $V_1, \ldots, V_n$, $A_1, \ldots, A_r$, $W_1, \ldots, W_s$ and $B_1, \ldots, B_t$ are id-terms.

$ScalarMethod$ and $SetMethod$ are id-terms. In the data-expression (a) and (b) from definition 5.2.2, T and S_i are id-terms that represent the output of the respective methods. For the signature expressions (c) and (d), A_i and B_j are id-terms that represent *types* of the results returned by the respective methods when they are invoked on an object of class C with arguments V_1, ..., V_n and $\mathrm{R}_1, \ldots, \mathrm{R}_l$ respectively.
Note that the same method or the same data/signature expression may have multiple occurrences in the same molecule.

5.2.3 Complex Formulas

Definition 5.2.3 gives the definition of F-formulea. F-formulas are built out of simpler F-formulas by means of logical connectives and quantifiers.

Definition 5.2.3 *(F-formulas)*

An F-formulae can be one of the following:

- *Molecular formulea are F-formulas;*
- $\varphi \vee \psi$*,* $\varphi \wedge \psi$*,* $\neg\varphi$ *are F-formulae if* φ *and* ψ *are F-formulae;*
- $\forall X\ \varphi$*,* $\exists Y\ \psi$ *are F-formulae when* φ *and* ψ *are F-formulae and* X *and* Y *are variables.*

Note that F-Logic also makes use of the implication connective "$\leftarrow$". The implication connective is defined in F-logic in the same way as in classical logic, namely $\varphi \leftarrow \psi$ is the same as $\varphi \vee \neg\psi$.

5.3 Path expressions

As we have seen, objects are directly accessible through their object names. Another way of accessing them is to navigate to them by applying a method to another object. This can be done by means of *pathexpressions*. Note that path expressions are not unique to F-logic, they are also used in many other object-oriented languages. For example, the object accessible via the object name `dinf` is also accessible by calling the method *dep* on the object `bill` when `bill[dep` $\rightarrow$ `dinf]` holds. The corresponding path expression for this is `"bill.dep"`. It is also possible that path expressions contain methods that take some arguments. Suppose that *vub* is a member of the *institution* class, thus `vub : institution` . Then we can refer to all publications written by bill at the vub by means of the path expression `"bill.papers@vub"`.
It is possible to nest path expressions at any position where id-temrs are allowed in F-molecules as well as P-molecules (predicates).

5.4 Semantics

In this section we will briefly introduce the semantics of F-logic. However, a complete description of the F-logic semantics lies out of the scope of this dissertation. Therefore we refer the interested reader for more details to [36].

In F-logic formulas are interpreted by a semantic structure, a so-called F-structure. Given an F-language $\mathcal{L}$ (see def. 5.2.1), an F-structure is a tuple $\mathbf{I} = < U, \prec_U, \in_U, I_{\mathcal{F}}, I_{\rightarrow}, I_{\twoheadrightarrow}, I_{\bullet\!\!\rightarrow}, I_{\bullet\!\!\twoheadrightarrow}, I_{\Rightarrow}, I_{\Rightarrow\!\!\!\!\Rightarrow} >$ for which:

- U is the domain of $\mathbf{I}$. This is similar to classical logic where U can be seen as a set of all actual objects in a possible world $\mathbf{I}$.
- $\prec_U$ is an irreflexive partial order on U denoting the semantic counterpart of the subclass-relationship. This means that $a \prec_U b$ is interpreted as a being a subclass of b.
- $\in_U$ is a binary relation which is used to model class membership. This means that $a \in_U b$ is interpreted as a being a member of class b.
- $I_{\mathcal{F}}$ denotes the mapping for interpreting function symbols. In F-logic this is done in the same way as in predicate calculus. This mapping thus interprets each k-ary object constructor $f \in \mathcal{F}$ by a function $U^k \rightarrow U$. For $k = 0$, $I_{\mathcal{F}}(f)$ can be identified with an element of the domain U.
- $I_{\rightarrow}$, $I_{\twoheadrightarrow}$, $I_{\bullet\!\!\rightarrow}$, $I_{\bullet\!\!\twoheadrightarrow}$, $I_{\Rightarrow}$, $I_{\Rightarrow\!\!\!\!\Rightarrow}$ denote mappings for interpreting each of the six types of method expressions in F-logic. For more details about these mappings we refer the reader to [36].

A variable assigment is a mapping from the set of variables $\mathcal{V}$ of an F-language $\mathcal{L}$ to the domain U. Such a variable assignment is denoted v. Take d to be a variable, if $d \in \mathcal{F}$ has arity 0 then $v(d) = I_{\mathcal{F}}(d)$. Recursively, $v(f(..., T, ...)) = I_{\mathcal{F}}(f)(..., v(T), ...)$.

Definition 5.4.1 *(Satisfaction of F-Molecules)*

Let $\boldsymbol{I}$ *be an F-structure and* G *be an F-molecule. We write* $I \models_v G$ *if and only if all of the following holds:*

- *When* G *is an is-a assertion then:*

(i) $v(Q) \preceq_U v(P)$, *if* $G = Q :: P$; *or*
$v(Q) \in_U v(P)$, *if* $G = Q : P$

- *When G is an object molecule of the form* ***O[a ';'-separated list of method expressions*** *] then for every method expression E in G, the following conditions must hold:*

(ii) *If E is a non-inheritable scalar data expression of the form* $ScalM@Q_1$, *...,* $Q_k \to T$*, the element* $I^{(k)}_{\to}$*(v(ScalM))(v(O), v(*Q_1*, ..., v(*$Q_(k)$*)))* *must be defined and equal v(T).*
Similar conditions must hold if E is an inheritable scalar data expression, except that $I^{(k)}_{\to}$ *should be replaced with* $I^{(k)}_{\bullet\to}$.

(iii) *If E is a non-inheritable set-valued data expression, of the form* $SetM@R_1$*, ...,* $R_l \twoheadrightarrow \{S_1, ..., S_m\}$*, the set* $I^{(l)}_{\twoheadrightarrow}$*(v(SetM))(v(O), v(*$R_1$*), ..., v(*$R_l$*)) must be defined and contain the set* $\{v(S_1), ..., v(S_m)\}$.
Similar conditions must hold if E is an inheritable set-valued data expression, except that $I^{(l)}_{\twoheadrightarrow}$ *should be replaced by* $I^{(l)}_{\bullet\twoheadrightarrow}$

(iv) *If E is a scalar signature expression,* $ScalM@Q_1$*, ...* $Q_n \Rightarrow (R_1$*, ...,* $R_u)$*, then the set* $I^{(n)}_{\Rightarrow}$*(v(ScalM))(v(O), v(*Q_1*, ..., v(*Q_n*)) must be defined and contain* $\{v(R_1), ..., v(R_u)\}$.

(v) *If E is a set-valued signature expression of the form* $SetM@V_1$*, ...,* $V_s \Rrightarrow (W_1, ..., W_v)$*, the set* $I^{(s)}_{\Rrightarrow}$*(v(SetM))(v(O), v(*V_1*, ..., v(*V_n*)) must be defined and contain* $\{v(W_1), ..., v(W_v)\}$.

Conditions (ii) and (iii) state that in case of a data-expression, the interpreting function must be defined on appropriate arguments and yield results compatible with those specified by the expression. Conditions (iv) and (v) state that in case of a signature expression, the type of a method specified by the expression must comply with the type assigned to this method by **I**.

The meaning of the formulaes $\varphi \vee \psi$, $\varphi \wedge \psi$ and $\neg\varphi$ are defined in the standard way:

- $\mathbf{I} \models_v \varphi \vee \psi \Leftrightarrow \mathbf{I} \models_v \varphi$ or $\mathbf{I} \models_v \psi$.
- $\mathbf{I} \models_v \varphi \wedge \psi \Leftrightarrow \mathbf{I} \models_v \varphi$ and $\mathbf{I} \models_v \psi$.
- $\mathbf{I} \models_v \neg\varphi \Leftrightarrow \mathbf{I} \not\models_v \varphi$

Note that the negation we will use in the formalization is denoted as *negation as failure*. This means that ¬p will be true when p cannot be derived as being true. This differs from the logical classical negation where ¬p will be undefined when p is not known. Here ¬p will be true when p is known to be false while ¬p will be false if p is known to be true.

Also note that for a closed formula[1] we can omit the mention of v and simply write $I \models \varphi$, since the meaning of a closed formula is independent of the choice of variable assignments.
An F-structure, **I**, is a *model* of a closed formula, ψ, if and only if $\mathbf{I} \models \psi$.

5.5 Why F-Logic?

To provide a formalization of the high-level modeling concepts of the VR-WISE approach we need a formal language that we can use for this formalization. As mentioned in chapter 3, the underlying representation formalism currently used by our approach is based on ontologies. More specific, we are using the Web Ontology Language OWL[2]. OWL, and more specifically the OWL DL variant (which is a sub-language of OWL). This is a notational variant of Description Logics (DL) [6]. DL are a family of knowledge representation formalisms that represent the knowledge of a domain by means of concepts in the domain and by using these concepts to specify properties of objects and individuals in the domain. Some examples of DL languages are $\mathcal{SHIF}$(D) and $\mathcal{SHOIN}$(D).
At the first sight, since our approach is following the object-oriented paradigm, DL may look suitable as the formalization language for our modeling concepts. However there is a major problem that restrains us from using DL as the formalization language. The problem is that DL doesn't allow meta-modeling.
The VR-WISE approach actually contains three levels. As we have seen in chapter 3, we have the conceptual domain specification, which contains all the concepts specified for the application domain and the relations between these concepts. Next, we have the conceptual world specification, which is an instantiation of the conceptual domain specification. However, there is a third level, which acts as a meta-level. This level contains our high-level modeling concepts used for the creation of the conceptual specifications.

[1] A closed formula does not contain free variables. A variable inside a formula is free when it is not bound by means of a quantifier.

[2] http://www.w3.org/TR/owl-features/

Figure 5.1 illustrates these three levels. Because we want to formalize the modeling concepts (the meta-level), we need to be able to use all three levels in our formal specifications.

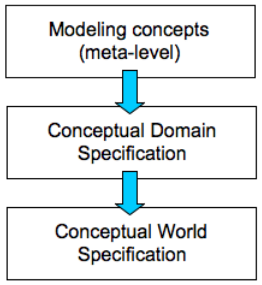

Modeling concepts (meta-level)
Conceptual Domain Specification
Conceptual World Specification

Figure 5.1: Three levels of formalization

We will look at an example. Suppose we specify a concept 'Airplane' at the conceptual domain specification. This concept is actually an instance of the modeling concept *Concept*. Suppose now that we create an instance 'Airbus380' in the conceptual world specification as an instance of the concept 'Airplane'. First 'Airplane' was an instance of *Concept* which now has to act as a class for the instance 'Airbus380'. As DL requires a strict separation between classes, individuals and properties it is not the most suited language to use for our purpose.

Classical logics, like predicate logic [64] are also not very suitable because they don't follow the object-oriented approach. To be able to use predicate logic we would need to extend this with object-oriented constructs. This is how we came to F-logic. F-logic is an object-oriented logic that can be seen as an extension of predicate logic.

When investigating the F-logic language it became clear that this logic language would fit the requirements we need for our formalization language. First of all, it supports the object-oriented paradigm. And next, classes can be treated as objects, which allows meta modeling. Although not crucial, but also an advantage, is the fact that we can query the data-level (domain and world specification) as well as the meta-level with the same language. This can be an extra advantage when one wants to perform intelligent reasoning.

Because of these reasons we have opted for F-logic as the language for the formalization. Note that other object-oriented logic languages exist. A few examples are C-logic [14], COL [3] and O-logic [35], each with its own strengths and weaknesses. However, among all of these approaches, F-Logic is the most elaborated one. Also note that F-logic borrowed some ideas from O-logic.
Also note that we could use OWL-Flight [17] as the underlying representation mechanism for the conceptual specifications. OWL-Flight is an ontology language for which the semantics has been based on F-Logic. To reflect the formalization of the modeling concepts presented in this chapter inside our implementation, all we need to do is to substitute OWL by OWL Flight.

5.6 Conclusion

In this chapter we have given a short overview of the frame-based language called Frame-Logic. We also provided the syntax of F-Logic and we explained path expressions in F-Logic. Next, we also motivated our choice of F-Logic as the formalization mechanism. This introduction provides the reader the necessary information for understanding the F-Logic constructs used in the rest of this dissertation. We refer the interested reader for more details on F-Logic to [36], [40] and [23].

Chapter 6

Fundamentals

In this section a number of general classes (such as point, orientation, ...) and deductive rules needed in rest of the formalization, will be defined.

6.1 Point

A point, in our conceptual modeling language for VR, is given by means of an x, y and z coordinate. The x, y and z coordinate are of the type float. Using the F-logic formalism we can define the modeling concept point as a class with three properties x, y and z. Definition 6.1.1 gives the definition of the class point using the F-logic formalism.

Definition 6.1.1 *A point is defined as*

point[x ⇒ float;
    y ⇒ float;
    z ⇒ float]

6.2 Orientation

Each VR object in a virtual environment has an orientation. Hence, in the VR-WISE approach, each object specified in the conceptual specifications gets a default orientation. This default orientation is illustrated in figure 6.1.
The default orientation of a concept can be changed by rotating the concept around on or more of the axes of the reference frame. In order to

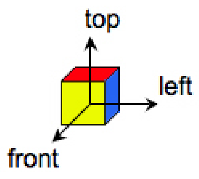


Figure 6.1: Default orientation of a concept in the VR-WISE approach

express the orientation of a concept, we define the class orientation with properties frontAngle, leftAngle and topAngle, each representing a rotation angle around respectively the front, left and top axis of the global reference frame. In the default situation all rotation angles are 0. This is reflected in definition 6.2.1.

Definition 6.2.1 *An orientation is defined as*

orientation[frontAngle ●→ 0;
leftAngle ●→ 0;
topAngle ●→ 0]

6.3 Rotations

In our formalization, rotations are defined by means of deductive rules. The rotation of a point with x, y and z coordinates around the axes of the local reference frame of an object, can be defined as three separate rotations around the three axes of the reference frame. Using mathematics, rotations can be written as 3×3 matrices.
The rotation around the X-axis with an angle α can be written as a 3×3 matrix as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\alpha) & -sin(\alpha) \\ 0 & sin(\alpha) & cos(\alpha) \end{bmatrix}$$

The rotation around the Y-axis with an angle β can be written as the following 3×3 matrix:

$$\begin{bmatrix} cos(\beta) & 0 & sin(\beta) \\ 0 & 1 & 0 \\ -sin(\beta) & 0 & cos(\beta) \end{bmatrix}$$

And finally the rotation around the Z-axis with an angle σ can be written as the 3×3 matrix:

$$\begin{bmatrix} cos(\sigma) & -sin(\sigma) & 0 \\ sin(\sigma) & cos(\sigma) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When combining these three rotations into one rotation matrix we get the 3×3 matrix:

$$\begin{bmatrix} cos(\beta)cos(\sigma) & -cos(\beta)sin(\sigma) & sin(\beta) \\ cos(\sigma)sin(\alpha)sin(\beta) + cos(\alpha)sin(\alpha) & cos(\alpha)cos(\sigma) - sin(\alpha)sin(\beta)sin(\sigma) & -cos(\beta)sin(\alpha) \\ -cos(\alpha)cos(\sigma)sin(\beta) + sin(\alpha)sin(\sigma) & cos(\sigma)sin(\alpha) + cos(\alpha)sin(\beta)sin(\sigma) & cos(\alpha)cos(\beta) \end{bmatrix}$$

Lets call the above matrix A. The matrix A is the matrix defining a combined rotation around the X-axis with α degrees, around the Y-axis with β degrees and around the Z-axis with σ degrees. Rotating a point x, y, z in this way can be written as

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = A \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Therefore, x', y', z' can be calculated using the following mathematical formulas:

$$x' = xcos(\beta)cos(\sigma) + zsin(\beta) - ycos(\beta)sin(\sigma)$$

$$y' = -zcos()\beta)sin(\alpha) + x[cos(\sigma)sin(\alpha)sin(\beta) + cos(\alpha)sin(\sigma)] + \\ y(cos(\alpha)cos(\sigma) - sin(\alpha)sin(\beta)sin(\sigma))$$

$$z' = zcos(\alpha)cos(\beta) + x[-cos(\alpha)cos(\sigma)sin(\beta) + sin(\alpha)sin(\sigma)] + \\ y[cos(\sigma)sin(\alpha) + cos(\alpha)sin(\beta)sin(\sigma)]$$

We will now define a deductive rule $rotate(P, \alpha, \beta, \sigma, P')$ as follows: given a point P, P' is the point which results after rotating the point P α degrees around the X-axis, β degrees around the Y-axis and σ degrees around the Z-axis.

To define this deductive rule, we decompose the problem in three other, more simple problems, and first define deductive rules called *xRotate*, *yRotate* and *zRotate* as follows. Given a point P, *xRotate* calculates X' as the new x-value of the resulting point when rotating the point P α degrees around the X-axis, β degrees around the Y-axis and σ degrees around the Z-axis. *yRotate* and *zRotate* do a similar calculation for the y-value and z-value respectively of the resulting point. Note the use of the operator 'is' inside the following definitions. This operator is used in Flora-2 [68] to evaluate arithmetic operators. The arithmetic operators like '+' and '$*$' are also provided inside Flora-2.

Definition 6.3.1

$$\begin{aligned}&\textit{xRotate(P, } \alpha, \beta, \sigma, \textit{X')} \leftarrow \textit{P : point} \wedge \textit{P[} x \rightarrow X, y \rightarrow Y, z \rightarrow Z] \wedge \\ &\qquad X' \textit{ is } X * cos(\beta) * cos(\sigma) + Z * sin(\beta) - Y * cos(\beta) * sin(\sigma)\end{aligned}$$

Definition 6.3.2

$$\begin{aligned}&\textit{yRotate(P, } \alpha, \beta, \sigma, \textit{Y')} \leftarrow \textit{P : point} \wedge \textit{P[} x \rightarrow X, y \rightarrow Y, z \rightarrow Z] \wedge \\ &\qquad Y' \textit{ is } -Z * cos(\beta) * sin(\alpha) + \\ &\qquad\qquad X * [cos(\sigma) * sin(\alpha) * sin(\beta) + cos(\alpha) * sin(\sigma)] + \\ &\qquad\qquad Y * [cos(\alpha) * cos(\sigma) - sin(\alpha) * sin(\beta) * sin(\sigma)]\end{aligned}$$

Definition 6.3.3

$$\begin{aligned}&\textit{zRotate(P, } \alpha, \beta, \sigma, \textit{Z')} \leftarrow \textit{P : point} \wedge \textit{P[} x \rightarrow X, y \rightarrow Y, z \rightarrow Z] \wedge \\ &\qquad Z' \textit{ is } Z * cos(\alpha) * cos(\beta) + \\ &\qquad\qquad X * [-cos(\alpha) * cos(\sigma) * sin(\beta) + sin(\alpha) * sin(\sigma)] + \\ &\qquad\qquad Y * [cos(\sigma) * sin(\alpha) + cos(\alpha) * sin(\beta) * sin(\sigma)]\end{aligned}$$

Now we can define the deductive rule $rotate(P, \alpha, \beta, \sigma, P')$ where the point P' is the result of rotating the point P α degrees around the X-axis, β degrees around the Y-axis and σ degrees around the Z-axis. This is given in definition 6.3.4.

Definition 6.3.4

$$\begin{aligned}\textit{rotate(P, } \alpha, \beta, \sigma, \textit{P')} \leftarrow\ & \textit{xRotate(P, } \alpha, \beta, \sigma, \textit{X')} \wedge \\ & \textit{yRotate(P, } \alpha, \beta, \sigma, \textit{Y')} \wedge \\ & \textit{zRotate(P, } \alpha, \beta, \sigma, \textit{Z')} \wedge \\ & \textit{P' : point[} x \rightarrow X', y \rightarrow Y', z \rightarrow Z']\end{aligned}$$

6.4 Other fundamentals

We will also need some mathematical objects inside our formalization. Some examples are a line or a circle. In this section we will define their F-logic variant.

Definition 6.4.1 *We formally define a line with the following parametric equations:*

$$\begin{cases} x = x_0 + ta \\ y = y_0 + tb \\ z = z_0 + tc \end{cases}$$

as follows in F-logic:

line [x_0 ⇒ float;
y_0 ⇒ float;
z_0 ⇒ float;
a ⇒ float;
b ⇒ float;
c ⇒ float]

Definition 6.4.2 *We formally define a vector (a, b, c) as follows in F-logic:*

vector [a ⇒ float;
b ⇒ float;
c ⇒ float]

Definition 6.4.3 *We formally define a circle with equation (x, y, z) = C + R* cos(*t*) *U + R* sin(*t*) *V as follows in F-logic:*

circle [c ⇒ point;
r ⇒ float;
u ⇒ vector;
v ⇒ vector]

Definition 6.4.4 *We formally define a cone as follows:*

cone [vertex ⇒ point;
height ⇒ float;
semiminor ⇒ float;
semimajor ⇒ float;
rotationAxis ⇒ vector;
rotationAngle ⇒ float]

So a cone has a vertex, a height and both a semimajor and semiminor axis. As we will see, all cones we will define open either along the X, Y or Z axis. To be able to define cones which open in another direction, we allow a rotation around a certain vector through the vertex. This is represented by the properties $rotationAxis$ and $rotationAngle$.

Chapter 7

Formalizing Static World Modeling Concepts

In this chapter we give the formal specifications for the modeling concepts inside the VR-WISE approach. These concepts were formally defined in chapter 3.

7.1 Concepts

As we have discussed in chapter 4, a concept has a name, a number of attributes which can have a default value, an orientation and a position. The position can either be exact or relative. When the designer specifies a position by means of a point (coordinates) we have an exact position for the concept. When the position of a concept is given by means of some spatial relation or connection relation (as we will see later) we say that the concept has a relative position. Also the orientation can be exact or relative. Remember that we are using a default orientation (see figure 6.1). To allow other orientations, we have introduced the notions of external and internal orientation (see chapter 3). Just as a reminder we will give the idea behind the external and internal orientation once more.

- **External orientation:** Using the notion of external orientation, we can indicate that the concept is rotated around some of the axes of its reference frame. This means that an instance of this concept will be rotated around some of the axes of the reference frame and this will be visible in the virtual environment. An external orientation rotates the complete concept in the virtual environment.

- **Internal orientation:** The notion of internal orientation is used to indicate that the concept is using a different left-right, top-bottom or front-back direction. An internal orientation is actually a rotation of the local reference frame of a concept around some of the axes of the global reference frame (which is equal to the default reference frame).

A concept from our conceptual model is modeled in F-Logic as a class. A class representing a concept needs to be a subclass of the class *concept*. The class *concept* is defined in definition 7.1.1.

Definition 7.1.1 *A concept is defined as*

$$\begin{aligned}&\textit{concept[position} \Rightarrow \textit{point;} && (1)\\ &\quad\textit{internalOrientation} \Rightarrow \textit{orientation;} && (2)\\ &\quad\textit{externalOrientation} \Rightarrow \textit{orientation]} && (3)\end{aligned}$$

The properties *position* (1), *internalOrientation* (2) and *externalOrientation* (3) are methods. We will define these methods when we discuss the definitions of spatial, orientation or connection relations later in this chapter.

We will also define a number of predicates that can be used to retrieve the front-, left- and top-angle of the internal orientation for a concept as well as the front-, left- and top-angle of the external orientation. All these predicates are similar. To illustrate them we define the predicate *iFrontAngle(A, α)* in definition 7.1.2. This predicate is true when α is the front angle of the internal orientation of a concept A.

Definition 7.1.2

$$\begin{aligned}\textit{iFrontAngle}(A,\ \alpha) \leftarrow\ & A : \textit{concept} \wedge\\ & A[\textit{internalOrientation} \rightarrow I] \wedge\\ & I[\textit{frontAngle} \rightarrow \alpha]\end{aligned}$$

It is easy to see that the predicates *iLeftAngle*, *iTopAngle*, *eFrontAngle*, *eLeftAngle* and *eTopAngle* can be defined similarly.

We will now look to an example. We take a car as a concept in our model. This means that car is defined as a subclass of *concept*.

car :: *concept*

The concept *car* may have one or more properties. Suppose in our case it has a *weight* property with default value 1500 and a *color* property with default value red. All properties of a concept will be defined as inheritable properties. This way, all subclasses and instances of a concept will inherit the default values. They can be overwritten if necessary for a specific subclass or instance. Lets go back to our example. The properties for *car* are defined as follows:

$$\begin{aligned}&car[weight \bullet\!\!\rightarrow 1500;\\ &\quad color \bullet\!\!\rightarrow red]\end{aligned}$$

Suppose now that we want to specify an exact position for the car. This means that we need to define a default position value for the *car* concept, which means that each car-instance will be positioned at this given position. Suppose this default position is the point (5,8,3). We write this down in the formalization as follows:

$$\begin{aligned}car[position \bullet\!\!\rightarrow point[&x \rightarrow 5;\\ &y \rightarrow 8;\\ &z \rightarrow 3]]\end{aligned}$$

7.2 Instances

An instance of a concept will be represented as the F-logic instance of the corresponding F-logic concept. Definition 7.2.1 gives the definition of the modeling concept instance.

Definition 7.2.1 *Let c be a concept, thus $c :: concept$, then an instance i of concept c is defined as $i : c$*

Lets go back to our example. Now suppose we have an instance *myCar*, which is an instance of the concept car defined earlier. This instance is defined as follows:

$myCar : car$

Since the properties of the concept *car* are defined as inheritable properties, the instance will inherit these default values. So the following statement is derivable:

$myCar[weight \rightarrow 1500]$

Suppose now that the $myCar$ instance needs to have a weight of 1830. To define this we can overwrite the default value as follows:
$myCar[weight \rightarrow 1830]$

In the next section we will look to the formalization of complex concepts and their corresponding instances.

7.3 Complex Concepts and Instances of Complex Concepts

Remember from chapter 4 that a complex concept consists of a number of parts. Parts can be simple concepts or complex concepts on their own. To define complex concepts, the $partOf$ property is used, which allows expressing that one concept is a part of another concept. Definition 7.3.1 defines how the part of relation is specified.

Definition 7.3.1 *Let a and b be two concepts, thus a::concept and b::concept. The fact that a is part of b is expressed by adding a property partOf to the concept definition of a:*
a[partOf ⇒ b]

Remember from chapter 4 that a complex concept also has a reference part. This means that the position and orientation of the reference part is also the position and orientation of the complex concept. All other parts of the complex concept can then be positioned and oriented relative to this reference part. This is expressed by the referencePartOf property (see definition 7.3.2).

Definition 7.3.2 *Let a and b be two concepts, thus a::concept and b::concept. The fact that a is the reference part of b is expressed by adding a property referencePartOf to the concept definition of a:*
$a[referencePartOf \Rightarrow b]$

Note that when a concept is the reference part of a complex concept, then by definition that concept should also be a part of this complex concept. This is expressed in definition 7.3.3.

Definition 7.3.3 *Let a and b be two concepts, thus a::concept and b::concept, where a is the reference part of b. Then the following deductive rule holds:*
$a[partOf \Rightarrow b] \leftarrow a[referencePartOf \Rightarrow b]$

The $partOf$ property and the $referencePartOf$ property now allow us to formalize the modeling concept *complexConcept*. Definition 7.3.4 gives the formalization of a complex concept. *complexConcept* can be defined as (1) a subclass of the concept class (since a complex concept is also a concept) with the properties (methods) $allParts$ (2) and $referencePart$ (3) returning respectively all the parts of the complex concept, and the reference part.

Definition 7.3.4 *The modeling concept complexConcept is defined as follows:*

$$\begin{array}{lr}
complexConcept :: concept & (1)\\
complexConcept[allParts \Rrightarrow concept; & (2)\\
\qquad referencePart \Rightarrow concept] & (3)
\end{array}$$

where allParts and referencePart are methods defined using the following deductive rules:

$$\begin{array}{l}
C[allParts \twoheadrightarrow A] \leftarrow C : complexConcept \wedge\\
\qquad A : concept \wedge\\
\qquad A[partOf \rightarrow C]
\end{array}$$

C[referencePart $\rightarrow$ *A]* $\leftarrow$ *C : complexConcept* $\wedge$
A : concept $\wedge$
A[referencePartOf $\rightarrow$ *C]*

Next we need to define the relationship between the position/orientation of a complex concept and the position/orientation of its reference part. Actually, when a complex concept has a reference part, then the position and orientation of an instance of the complex concept will be the same as respectively the position and orientation of the reference part.

Definition 7.3.5 gives the definition of the referenced position predicate. Given three attributes, A, C, and P, this predicate is true when A is an instance of a complex concept, C is an instance of a concept and is the reference part of A, and P is an instance of point and it is the position of the instance C.

Definition 7.3.5
referencedPosition(A, C, P) $\leftarrow$ *A : complexConcept* $\wedge$
C : concept $\wedge$ *P : point* $\wedge$
C[referencePartOf $\rightarrow$ *A]* $\wedge$
C[position $\rightarrow$ *P]*

The $referencedPosition$ predicate allows us to continue with the definition of *concept* (see definition 7.1.1). We have already mentioned that the *position* property is a method, but we did not yet define this method. Definition 7.3.6 gives the definition for the *position* method for a *concept*.

Definition 7.3.6
A[position $\rightarrow$ *P]* $\leftarrow$ *A:concept* $\wedge$ *C:concept* $\wedge$ *P:point* $\wedge$
referencedPosition(A, C, P)

So when A and C are instances of a concept and P is an instance of point, and the predicate $referencedPosition(A, C, P)$ is true (which means that P is the position of C which is the reference part of A), then the *position* method will return P as value.

Now lets look to an example. Suppose we define a car as a complex concept. Suppose that chassis and dashboard are two parts of the car concept and that the chassis is the reference part of the complex concept. In our formalization this is represented as follows:

$car :: complexConcept$
$chassis :: concept$
$dashboard :: concept$
$dashboard[partOf \Rightarrow car]$
$chassis[referencePartOf \Rightarrow car]$

Note that $chassis[partOf \Rightarrow car]$ is implicit because of definition 7.3.3. So far our example defined the concept level of our conceptual specification. Now suppose we create an instance myCar of car, an instance myDashboard of the concept dashboard and an instance myChassis of chassis. Suppose also that the instances myDashboard and myChassis are part of myCar and that myChassis is the reference part of myCar. This is formally specified as follows:

$myCar : car$
$myChassis : chassis$
$myDashboard : dashboard$
$myDashboard[partOf \rightarrow myCar]$
$myChassis[referencePartOf \rightarrow myCar]$

The above formalization defines the instance level of our model. If we would now like to know the position of the instance myCar in the virtual environment we could use the *position* method from the concept class. Tools like OntoBroker or Flora-2 will then try to resolve the predicate $referencedPosition(myCar, C, P)$. This would result in the following:

$$\begin{aligned} \text{referencedPosition(myCar, C, P)} \leftarrow\ & \text{myCar : complexConcept} \wedge \\ & \text{C : concept} \wedge \text{P : point} \wedge \\ & \text{C[referencePartOf} \rightarrow \text{myCar)} \wedge \\ & \text{C[position} \rightarrow \text{P]} \end{aligned}$$

The predicate can be resolved when C is $myChassis$ since $myChassis[referencePartOf \rightarrow myCar]$ holds. So the position P is the position of the instance $myChassis$.

So far we have defined the relative position of a complex concept according to its reference part. Now similar we will define the relative orientation of a complex concept with respect to its reference part. To do this we will first define two predicates $referencedExtOrientation$ and $referencedIntOrientation$. Definition 7.3.7 gives the definition of the $referencedExtOrientation$ predicate.

Definition 7.3.7
referencedExtOrientation(A, C, E) ← *A : complexConcept* ∧
C : concept ∧ *E : orientation* ∧
C[referencePartOf → A] ∧
C[externalOrientation → E]

So when C is an instance of a concept being the reference part of an instance A of a complex concept and E is an orientation being the external orientation of C, then the predicate $referencedExtOrientation(A, C, E)$ is true.

Again, the $referencedExtOrientation$ predicate allows us to continue with the *concept* definition (def. 7.1.1). More specific, we can now define the signature of the *externalOrientation* method. The definition of the *externalOrientation* method is given in definition 7.3.8.

Definition 7.3.8
A[externalOrientation → E] ← *A:concept* ∧ *C: concept* ∧
E:orientation ∧
referencedExtOrientation(A, C, E)

So when A and C are instances of concepts and E is an orientation and the predicate $referencedExtOrientation(A, C, E)$ is true (which means that E is the external orientation of C which is the reference part of A), then E is the value for the *externalOrientation* property of A. With other words, when E is the external orientation of C and C is the reference part of A, then E is the external orientation of A.
Similar we will define the semantics of the internal orientation of a complex concept. Definition 7.3.9 gives the definition of the $referencedIntOrientation$ predicate.

Definition 7.3.9
referencedIntOrientation(A, C, I) ← *A : complexConcept* ∧
C : concept ∧ *I : orientation* ∧
C[referencePartOf → A] ∧
C[internalOrientation → I]

Definition 7.3.9 states that when C is an instance of a concept being the reference part of an instance A of a complex concept and I is an orientation being the internal orientation of C, then the predicate $referencedIntOrientation(A, C, I)$ is true. The $referencedIntOrientation$ predicate allows us now to define the *internalOrientation* method for a concept (as used in definition 7.1.1). The *internalOrientation* method is defined in definition 7.3.10.

Definition 7.3.10

$$\begin{aligned} A[internalOrientation \rightarrow I] \leftarrow\ & A{:}concept \wedge C{:}concept \wedge \\ & I{:}orientation \wedge \\ & referencedIntOrientation(A,\ C,\ I) \end{aligned}$$

Definition 7.3.10 states that when C is the reference part of A and I is the internal orientation of C, then I is also the internal orientation of A.

So far we have seen how complex concepts are formalized. Note that all methods are working on instances of complex concepts. However, it may be interesting to be able to query the system about the concepts that are part of some complex concept. E.g., instead of asking for the instances which are part of a particular instance of the complex concept *car*, it may also be necessary to ask for the concepts that are part of the complex concept *car*. Therefore we need to overwrite the methods *allParts* and *referencePart* so they can also work with classes.

Definition 7.3.11

$$\begin{aligned} C[allParts \twoheadrightarrow A] \leftarrow & C :: complexConcept \wedge \\ & A :: concept \wedge \\ & A[partOf \Rightarrow C] \end{aligned}$$

$$\begin{aligned} C[referencePart \rightarrow Y] \leftarrow & C :: complexConcept \wedge \\ & A :: concept \wedge \\ & A[referencePartOf \Rightarrow C] \end{aligned}$$

Note that the definition for the methods *allParts* and *referencePart* in definition 7.3.11 is very similar to the one of definition 7.3.4. The only difference is that C now needs to be a subclass of the complexConcept concept instead of an instance and that A is a subclass of concept instead of an instance of concept. Since concepts are not represented in the virtual environment, they have no position or orientation (except for a possible default value for their instances). Therefore, we don't need to overload the $referencedPosition$, $referencedExtOrientation$ and $referencedIntOrientation$ for subclasses of concepts.

7.4 Roles

In chapter 4 we have introduced the concept *role* to be able indicating that a concept is playing a particular role in the context in which it is used. The context in which it is used is for instance a complex object. For example, inside a complex concept *car* we have the concept *wheel* playing the role $leftFrontWheel$. Here, the context in which the concept is playing its role is the complex concept car. So the role doesn't need to keep track of its context since it can be derived. Therefore we can define the modeling concept role as a subclass of a concept having one extra property, namely the name of the concept playing the role. Definition 7.4.1 defines this.

Definition 7.4.1 *A role is defined as a subclass of a concept having the additional property conceptName of type concept indicating the concept playing the role:*

role :: concept
role[conceptName ⇒ concept]

Since *role* is formalized as a subclass of concept it also inherits all methods of the concept class. This way the complete theory about part-of relations from the previous section also holds for roles. Roles can also be used in spatial relations, orientation relations, connection relations, etc.
Similar as with the modeling concept *concept*, a role in a conceptual model will be represented as a subclass of the modeling concept *role*. Take the following example where *wheel* is a concept and $leftFrontWheel$ is a role played by the concept $wheel$. This is formally defined as follows:

$wheel :: concept$
$leftFrontWheel :: role$
$leftFrontWheel[conceptName \Rightarrow wheel]$

We will now look to the instantiation of a role. Suppose $myLeftFrontWheel$ is an instance of the role $leftFrontWheel$. As we have seen in an informal way in chapter 4, $myLeftFrontWheel$ is an instance of the concept $wheel$ because it is an instance of the role $leftFrontWheel$ played by the concept $wheel$. This rule is expressed in definition 7.4.2.

Definition 7.4.2
X:B $\leftarrow$ *A::role* $\wedge$ *X:A* $\wedge$
A[conceptName $\Rightarrow$ *B]*

So definition 7.4.2 states that when A is a role, X is an instance of A and B is the concept playing the role A, then X is also an instance of B. Lets go back to our example. We create the instance $myLeftFrontWheel$ as follows:

$myLeftFrontWheel : leftFrontWheel$

If we ask our knowledge base for the concept(s) of which myLeftFrontWheel is an instance we will get the answer $wheel$. The query would look as follows:

$? - myLeftFrontWheel : X \wedge X :: concept$

The above query will return $X = wheel$ which is exactly the result we want.

7.5 Spatial Relations

In this section we will give a formalization of the spatial relations. As we have seen, spatial relations can be used to position an object relative to another object (see chapter 3). They can be applied on concepts as well as on instances. However they can also be used in the context of complex concepts on concepts, on instances, and on roles.
As explained in chapter 3, we have a number of different spatial relations. First, we have six elementary spatial relations, namely leftOf, rightOf, topOf, bottomOf, frontOf and backOf. Next, we have relations that are a combination of two elementary spatial relations, namely leftTopOf, leftBottomOf, frontLeftOf, backLeftOf, rightTopOf, rightBottomOf, frontRightOf, backRightOf, frontTopOf, backTopOf, frontBottomOf and backBottomOf. Finally, we have the relations that are a combination of three elementary spatial relations. These are frontLeftTopOf, backLeftTopOf, frontLeftBottomOf, backLeftBottomOf, frontRightTopOf, backRightTopOf, frontRightBottomOf and backLeftBottomOf. The definition of a spatial relation is given in definition 7.5.1.

Definition 7.5.1
spatialRelation[direction ⇒ string;
distance ⇒ float]

A spatial relation has thus two properties, namely *direction* and *distance*. Now we will define how to specify that one concept is positioned relative to another concept by means of a spatial relation. This is defined in definition 7.5.2. Note that the *positioned* property is defined as an inheritable property which means that each instance of a will be positioned by default to an instance b by means of the spatial relation r.

Definition 7.5.2 *Let a and b be two concepts, thus a :: concept and b :: concept. The fact that a is positioned relative to b with a distance d towards direction 'dir' is formally expressed as follows:*

r : spatialRelation[direction → 'dir';
distance → d]

a [positioned@b •→ r]

So far we have defined the declaration for the spatial relation. Next, we also need to define the semantics of the spatial relation. Therefore in the following paragraphs we will introduce the necessary definitions.

We will define a predicate $relativePosition(A, X, Y, Z, P)$. This predicate will be true when P is a point and A is a concept for which the following hold:

- the value of the property x of P equals X plus the x-value of the position of A, and,
- the value of the property y of P equals Y plus the y-value of the position of A, and,
- the value of the property z of P equals Z plus the z-value of the position of A.

Actually the predicate $relativePosition(A, X, Y, Z, P)$ is true when P is a point which lies X units left of A, Y units left of A and Z units top of A. This is because, as stated earlier, by definition the front direction corresponds to the x-direction, the left direction to the y-direction and the top direction to the z-direction. The predicate $relativePosition(A, X, Y, Z, P)$ is given in definition 7.5.3. The explaination is given afterwards.

Definition 7.5.3
relativePosition(A, X, Y, Z, P) $\leftarrow$

A : concept $\wedge$	*(1)*
iFrontAngle(A, iα) $\wedge$ *iLeftAngle(A, iβ)* $\wedge$	*(2)*
iTopAngle(A, iσ) $\wedge$ *eFrontAngle(A, eα)* $\wedge$	*(3)*
eLeftAngle(A, eβ) $\wedge$ *eTopAngle(A, eσ)* $\wedge$	*(4)*
P_1 *: point[x* $\rightarrow$ *X, y* $\rightarrow$ *Y, z* $\rightarrow$ *Z]* $\wedge$	*(6)*
rotate(P_1*, iα, iβ, iσ,* P_2*)* $\wedge$	*(7)*
rotate(P_2*, eα, eβ, eσ,* P_3*)* $\wedge$	*(8)*
A[position $\rightarrow$ P_A*]*	*(9)*
P : point[x $\rightarrow$ $P_3.x + P_A.x$*, y* $\rightarrow$ $P_3.y + P_A.y$*, z* $\rightarrow$ $P_3.z + P_A.z$*]*	*(10)*

We will now explain the above predicate in more detail. Let's assume that A is positioned at (0, 0, 0). Then P_1 = (X, Y, Z) (see line (6)) lies X units in front of A, Y units right of A and Z units above A. However, we also need to take into account the internal and external orientation. The predicate *rotate* in line (7) is true when P_2 is equal to the point P_1 after rotating it using the internal orientation angles of A. Next, the predicate *rotate* in line (8) is true when P_3 is equal to the point P_2 rotated using the external orientation angles of A. So P_3 needs to be the resulting point when rotating P_1 over the internal as well as the external orientation of A. Remember that we started with the assumption that A was positioned at (0, 0, 0). However, A can be at another position in the virtual environment, let's say P_A = (X_A, Y_A, Z_A) (see line (9)). So finally (line (10)) the *relativePosition* predicate is true when P lies X units in front of A, Y units right of A and Z units above A taking into account the internal and external orientation of A.

We can now use the *relativePosition* predicate to define the actual semantics of the spatial relations. We will start by defining the predicate $leftOfPos$ in definition 7.5.4. This predicate is true when P is a point which lies D units to the left of an instance C. This is actually the case when the predicate $relativePosition(C, 0, D, 0, P)$ is true.

Definition 7.5.4
leftOfPos(C, D, P) $\leftarrow$
C : concept $\wedge$ *P : point* $\wedge$
relativePosition(C, 0, D, 0, P)

We can define similar predicates *frontOfPos(C, D, P)*, *rightOfPos(C, D, P)*, *bottomOfPos(C, D, P)*, ... For example, *rightOfPos(C, D, P)* would be true when P is a point that lies D units right of an instance A. This definition is given in definition 7.5.5.

Definition 7.5.5
rightOfPos(C, D, P) $\leftarrow$
C : concept $\wedge$ *P : point* $\wedge$
relativePosition(C, 0, -D, 0, P)

Next we also need to formalize the meaning of spatial relations that are a combination of two or three simple spatial relations. Therefore we define the $leftTopOfPos$, $leftBottomOf$, $frontLeftTopOfPos$, ... predicates. We will start with the $frontLeftOfPos$ predicate. When the distance towards the front-left direction is D then we calculate the corresponding distances in the left direction and the front direction. Figure 7.1 illustrates this.

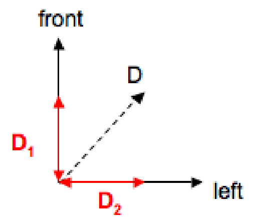


Figure 7.1: Distances in separate directions calculated from a combined direction

We need to calculate the distances D$_1$ and D$_2$. Note that we assume that the angle between D$_1$ and D and the angle between D$_2$ and D are equal. In principle, it is possible to have a distance D towards the front-left direction for which this is not the case. However, the $frontLeftOfPos$ relation is intended to be an intuitive modeling concept usable by people with little to no mathematical background. Therefore preciseness is sacrificed. Due to this assumption we can apply the rule of Pythagoras which says that $D = \sqrt{D_1^2 + D_2^2}$. Since a combination of two directions is always exactly in the middle of the two directions $D_1 = D_2$. Thus, $D = \sqrt{2}D_1$. Thus $D_1 = D_2 = \frac{D}{\sqrt{2}}$. Therefore, if an object A is frontLeftOf an object B with a distance D, this means that A is $\frac{D}{\sqrt{2}}$ units left of B and also $\frac{D}{\sqrt{2}}$ units front of B. This means $\frac{D}{\sqrt{2}}$ units in the X-direction (front) as well as in the Y-direction (left). This is reflected in the definition of the $frontLeftOfPos$ predicate given in definition 7.5.6.

Definition 7.5.6
frontLeftOfPos(C, D, P)$\leftarrow$ *C:Concept* $\wedge$ *P:Point* $\wedge$
relativePosition(C, $\frac{D}{\sqrt{2}}$, $\frac{D}{\sqrt{2}}$, *0, P)*

The other combinations are very similar. For example leftTopOfPos is formalized as follows:

Definition 7.5.7
leftTopOfPos(C, D, P)$\leftarrow$ *C:Concept* $\wedge$ *P:Point* $\wedge$
relativePosition(C, 0, $\frac{D}{\sqrt{2}}$, $\frac{D}{\sqrt{2}}$, *P)*

The formalization of a combination of three directions is similar. For example, we formalize the relation $frontLeftTopOfPos$ predicate as follows:

Definition 7.5.8
frontLeftTopOfPos(C, D, P)$\leftarrow$ *C:Concept* $\wedge$ *P:Point* $\wedge$
relativePosition(C, $\frac{D}{\sqrt{2}}$, $\frac{D}{\sqrt{2}}$, $\frac{D}{\sqrt{2}}$, *P)*

Next we define a predicate *spatialPosition(A, C, P)*, which is true when P is the position of an instance A which has a spatial relation with an instance C. To define this predicate we will make use of predicates defined earlier. The *spatialPosition(A, C, P)* predicate is given in definition 7.5.9.

Definition 7.5.9
spatialPosition(A, C, P) $\leftarrow$ *A : concept* $\wedge$
C : concept $\wedge$ *P : point* $\wedge$
R : spatialRelation $\wedge$ *A[positioned@C* $\rightarrow$ *R]* $\wedge$
R[distance $\rightarrow$ *D]* $\wedge$
(R[direction $\rightarrow$ *"leftOf"]* $\wedge$ *leftOfPos(C, D, P))* $\vee$ *(1)*
(R[direction $\rightarrow$ *"rightOf"]* $\wedge$ *rightOfPos(C, D, P))* $\vee$ *(2)*
...
(R[direction $\rightarrow$ *"leftTopOf"]* $\wedge$ *leftTopOfPos(C, D, P))* $\vee$ *(3)*
...
(R[direction $\rightarrow$ *"leftTopFrontOf"]* $\wedge$ *leftTopFrontOfPos(C, D, P)))]*

So *spatialPosition(A, C, P)* is true when for example the instance A is left of the instance C with a distance D and P is the point which lies D units left of instance C (see (1)). Hence, A is positioned at the point P which lies D units left of C. Or the *spatialPosition(A, C, P)* predicate can also be true when A is right of C with distance D and P is a point which lies rightOf C with distance D (see (2)). So here A is positioned at the point P which lies D units right of C. We can continue with all other possible spatial relations. Finally, we can use the *spatialPosition* predicate in order to refine the signature for the *position* method we already defined for *concept* in definition 7.3.6. We will now refine this method in such a way that when for a concept a spatial relation with another concept has been given, then the *position* method returns the relative position according to this spatial relation. The refined definition of the *position* method is given in definition 7.5.10.

Definition 7.5.10

$$\begin{aligned} &A[position \rightarrow P] \leftarrow A{:}concept \wedge C{:}concept \wedge P{:}point \wedge \\ &\qquad (referencedPosition(A,\ C,\ P) \vee && (1)\\ &\qquad spatialPosition(A,\ C,\ P)) && (2) \end{aligned}$$

So the *position* method for an instance A returns a point P such that either:

(1) A is an instance of a complex concept that has a reference part C position and point P. Hence P is also the position for A. Or,

(2) there is somewhere an instance C which is related to instance A via a spatial relation. Hence P is the position of A relative to C.

Now we will illustrate how the theory discussed above works in practice.

7.5.1 Example

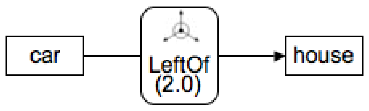


Figure 7.2: A spatial relation between two concepts

Take two concepts, car and house, where car is positioned 2 units left of house (see figure 7.2). This is formally expressed as:

$car :: concept$
$house :: concept$
$r : spatialRelation[direction \rightarrow "leftOf"; distance \rightarrow 2]$
$car[positioned@house \bullet\!\!\rightarrow r]$

Suppose also that we have an instance myCar of car and an instance my-House of house such that:

$myCar : car$
$myHouse : house$
$myCar[positioned@myHouse \rightarrow r]$

Systems like OntoBroker or Flora-2 can be used to query the system for the position of *myCar*. This query looks as follows:

$? - myCar[position \rightarrow P]$

The system will then start a number of deductions in order to resolve this query. First it starts with the *position* method:

$$\begin{aligned} myCar[position \rightarrow P] \leftarrow & myCar : concept \wedge C : concept \wedge P : point \wedge \\ & (referencedPosition(myCar, C, P) \vee \\ & spatialPosition(myCar, C, P)) \end{aligned}$$

At some point during the solving process the system will try to find an instance C and point P so that the predicate *spatialPosition(myCar, C, P)* becomes true. This means that C is spatially related to *myCar* and *myCar* is position at point P relative to C according to the spatial relation between *myCar* and C. So the *spatialPosition* predicate looks as follows:

$$\begin{aligned} & spatialPosition(myCar, C, P) \leftarrow myCar : concept \wedge \\ & \quad C : concept \wedge P : point \wedge \\ & \quad R : spatialRelation \wedge myCar[positioned@myHouse \rightarrow R] \wedge \\ & \quad R[distance \rightarrow D] \wedge \\ & \quad (R[direction \rightarrow "leftOf"] \wedge leftOfPos(C, D, P)) \vee \\ & \quad (R[direction \rightarrow "rightOf"] \wedge rightOfPos(C, D, P)) \vee \\ & \quad \ldots \\ & \quad (R[direction \rightarrow "leftTopOf"] \wedge leftTopOfPos(C, D, P)) \vee \\ & \quad \ldots \\ & \quad (R[direction \rightarrow "leftTopFrontOf"] \wedge leftTopFrontOfPos(C, D, P)))] \end{aligned}$$

Since there is a spatial relation instance r so that *myCar[positioned@myHouse → r]*, R will be substituted by r. Because of the definition of r, we know that *r[direction → "leftOf"]* holds. The branche *(R[direction → "leftOf"] ∧ leftOfPos(C, D, P))* will become *(r[direction → "leftOf"] ∧ leftOfPos(myHouse, 2, P))* to the system. The system will now try to find a point P so that $leftOfPos(myHouse, 2, P)$ is true. The *leftOfPos* predicate looks as follows:

$$
\begin{aligned}
&leftOfPos(myHouse, 2, P) \leftarrow \\
&\qquad myHouse : concept \wedge P : point \wedge \\
&\qquad relativePosition(myHouse, 0, 2, 0, P)
\end{aligned}
$$

Following the $leftOfPos$ predicate, the system will try to resolve the predicate $relativePosition(myHouse, 0, 2, 0, P)$. This results in

$$
\begin{aligned}
&relativePosition(myHouse, 0, 2, 0, P) \leftarrow \\
&\quad myHouse : concept \wedge P : point \wedge \\
&\quad iFrontAngle(myHouse, i\alpha) \wedge iLeftAngle(myHouse, i\beta) \wedge \\
&\quad iTopAngle(myHouse, i\sigma) \wedge eFrontAngle(myHouse, e\alpha) \wedge \\
&\quad eLeftAngle(myHouse, e\beta) \wedge eTopAngle(myHouse, e\sigma) \wedge \\
&\quad P_1 : point[x \leftarrow 0, y \leftarrow 2, z \leftarrow 0] \wedge \\
&\quad rotate(P_1, i\alpha, i\beta, i\sigma, P_2) \wedge \\
&\quad rotate(P_2, e\alpha, e\beta, e\sigma, P_3) \wedge \\
&\quad myHouse[position \rightarrow P_A] \\
&\quad P : point[x \rightarrow P_3.x + P_A.x, y \rightarrow P_3.y + P_A.y, z \rightarrow P_3.z + P_A.z]
\end{aligned}
$$

Now suppose that the instance *myHouse* has the default orientation. This means that $i\alpha = i\beta = i\sigma = e\alpha = e\beta = e\sigma = 0$. Suppose also that *myHouse* is positioned at (2, 3, 2), hence $X_A = 2$, $Y_A = 3$ and $Z_A = 1$. By the definition of the *rotate* predicate (see section 6.3) we will get $P[x \rightarrow 0 + 2, y \rightarrow 2 + 3, z \rightarrow 0 + 1]$. The position of myCar, P, will be $P[x \rightarrow 2, y \rightarrow 5, z \rightarrow 1]$. So the position of myCar in the virtual environment is $(2, 5, 1)$ while the position of myHouse is $(2, 3, 1)$. Since the global Y-axis represents the left direction by default we can see that the position of myCar is indeed 2 units left of myHouse in the virtual environment.
In this section we have formally defined the spatial relations. In the next section we will give formal definitions for the orientation relations.

7.6 Orientation Relations

In this section we will give formal definitions for the orientation relations. First we will give the formalization of relative orientation relations, which are orientation relations between two concepts, instances or roles. Next we will give the formalization for the orientation by angle relation.

7.6.1 Relative orientation relations

We will start with formally defining the relative orientation relations. As we have seen in chaper 3, we have simple relative orientation relations like frontToFront, frontToRight, topToFront, etc. Next, we also have combined relative orientation relations like frontToLeftTop, frontToFrontLeftTop, etc. Similar as we have done for the spatial relations, we will start by formally defining the declarative side of the orientation relations. The definition of an orientation relation is given in definition 7.6.1.

Definition 7.6.1
relativeOrientationRelation[sourceFace ⇒ string;
targetFace ⇒ string]

An orientation relation has thus two properties, namely $sourceFace$ and $targetFace$. If one wants to specify that the source is oriented with its front towards the front of the target, then the values for both properties become 'front'. Now we will define how to specify that one concept is oriented relative to another concept by means of an orientation relation. This is defined in definition 7.6.2. Note that the *oriented* property is defined as an inheritable property which means that each instance of a will be oriented by default to an instance b by means of the orientation relation r.

Definition 7.6.2 *Let a and b be two concepts, thus a :: concept and b :: concept. The fact that a is oriented with side 'aSide' relative to the 'bSide' of b is formally expressed as follows:*

r : relativeOrientationRelation[sourceFace → 'aSide';
targetFace → 'bSide']

a [oriented@b •→ r]

Note that the values of the *sourceFace* and *targetFace* properties can have as values all possible directions (front, back, . . .) or any valid combination of directions. All other values will just be ignored.

In the following paragraphs we will define the semantics of the orientation relations. We will start with a simple relative orientation relation. Take a and b to be two concepts where concept a needs to be oriented with its front towards the front of concept b. This is expressed graphically in figure 7.3.

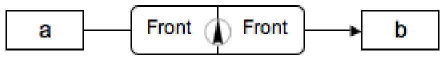


Figure 7.3: Example of a relative orientation relation between concepts

Now suppose we have two instances named $instanceA$ and $instanceB$. $InstanceA$ must be oriented with its front side to the front side of $instanceB$. This default situation is shown in figure 7.4.

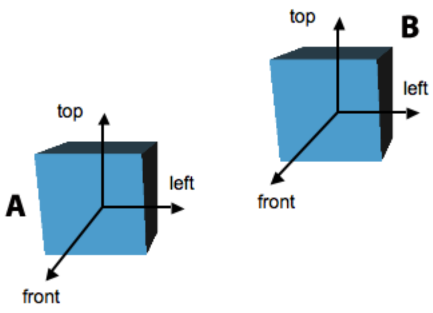


Figure 7.4: Default orientation of two objects

For each side of an object we can calculate the vector representing this side. Take for example the front side. We know that in the default orientation of an object, the front side corresponds to the x-axis of the local reference frame of the object. So the vector $(1, 0, 0)$ represents the front side of the object. However, we need also to take into account the internal and external orientation of the object. Therefore we need to rotate the vector representing the side around the object's internal and external orientation.
Now, definition 7.6.3 defines the *directionVectors(A, B, V_1, V_2)* predicate. This predicate is true when V_1 and V_2 are the vectors representing the side of A and the side of B respectively when A and B are oriented with repect to each other by means of a relative orientation relation.

Definition 7.6.3

$$
\begin{array}{lr}
\mathit{directionVectors}(A,\ B,\ V_1,\ V_2) \leftarrow A : \mathit{concept} \wedge & \\
\qquad B : \mathit{concept} \wedge A[\mathit{oriented}@B \bullet\!\!\rightarrow R] \wedge & \\
\qquad \mathit{iFrontAngle}(A,\ iA\alpha) \wedge \mathit{iLeftAngle}(A,\ iA\beta) \wedge & \\
\qquad \mathit{iTopAngle}(A,\ iA\sigma) \wedge \mathit{iFrontAngle}(B,\ iB\alpha) \wedge & \\
\qquad \mathit{iLeftAngle}(B,\ iB\beta) \wedge \mathit{iTopAngle}(B,\ iB\sigma) \wedge & \\
\qquad \mathit{eFrontAngle}(A,\ iA\alpha) \wedge \mathit{eLeftAngle}(A,\ iA\beta) \wedge & \\
\qquad \mathit{eTopAngle}(A,\ iA\sigma) \wedge \mathit{eFrontAngle}(B,\ iB\alpha) \wedge & \\
\qquad \mathit{eLeftAngle}(B,\ iB\beta) \wedge \mathit{eTopAngle}(B,\ iB\sigma) \wedge & \\
\qquad (\ (R[\mathit{sourceFace} \rightarrow \text{"front"}] \wedge X : \mathit{point}[x \rightarrow 1, & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y \rightarrow 0, & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad z \rightarrow 0]) \vee & \\
\qquad\quad \ldots & \\
\qquad\quad (R[\mathit{sourceFace} \rightarrow \text{"frontRight"}] \wedge X : \mathit{point}[x \rightarrow 1, & (1) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y \rightarrow -1, & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad z \rightarrow 0])) \wedge & \\
\qquad (\ (R[\mathit{targetFace} \rightarrow \text{"front"}] \wedge Y : \mathit{point}[x \rightarrow 1, & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y \rightarrow 0, & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad z \rightarrow 0]) \vee & \\
\qquad\quad \ldots & \\
\qquad\quad (R[\mathit{targetFace} \rightarrow \text{"frontRight"}] \wedge Y : \mathit{point}[x \rightarrow 1, & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad y \rightarrow -1, & \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad z \rightarrow 0])) \wedge & \\
\qquad \mathit{rotate}(X,\ iA\alpha,\ iA\beta,\ iA\sigma,\ X_1) \wedge & (2) \\
\qquad \mathit{rotate}(X_1,\ eA\alpha,\ eA\beta,\ eA\sigma,\ X_2) \wedge & (3) \\
\qquad \mathit{rotate}(Y,\ iB\alpha,\ iB\beta,\ iB\sigma,\ Y_1) \wedge & \\
\qquad \mathit{rotate}(Y_1,\ eB\alpha,\ eB\beta,\ eB\sigma,\ Y_2) \wedge & \\
\qquad V_1 : \mathit{vector}[a \rightarrow X_2.x,\ b \rightarrow X_2.y,\ c \rightarrow X_2.z] \wedge & (4) \\
\qquad V_2 : \mathit{vector}[a \rightarrow Y_2.x,\ b \rightarrow Y_2.y,\ c \rightarrow Y_2.z] &
\end{array}
$$

So, suppose that the face for the source side (thus for A) is equal to ”frontRight” then we define a point $(1, 0, 0)$. This is done in line (1). Next we rotate this point around the internal and external orientation of A (see lines (2)-(3)). Finally we turn the resulting point into a vector V_1 representing the face used on the source side of the relation (see line (4)). The same is done for the face used on the target side.
Now that we know the vectors representing the directions, we can define the semantics of the relative orientation relation. A relative orientation relation is fullfilled when the vectors representing the directions used in the relation have an opposite direction. Having an opposite direction thus means that

the angle between both vectors must be -180 degrees. Therefore, we need to calculate the angle between the vectors V$_1$ and V$_2$. Take $V_1 = (x_1, y_1, z_1)$ and $V_2 = (x_2, y_2, z_2)$. Then the angle between these vectors can be calculated using the following mathematical formula:

$$\cos\alpha = \frac{V_1 \cdot V_2}{||V_1||||V_2||}$$

Since the dot product of V_1 and V_2 equals $x_1x_2 + y_1y_2 + z_1z_2$ and the norm of a vector $V = (x, y, z)$ equals $\sqrt{x^2 + y^2 + z^2}$ we know that we can calculate the angle α as:

$$\cos\alpha = \frac{x_1x_2+y_1y_2+z_1z_2}{\sqrt{x_1^2+y_1^2+z_1^2}\sqrt{x_2^2+y_2^2+z_2^2}}$$

Therefore we define the predicate *angleBetween(*V_1*,* V_2*,* α*)* which is true when α is the angle between V$_1$ and V$_2$. This predicate is defined in definition 7.6.4.

Definition 7.6.4

angleBetween(V_1*,* V_2*,* α*)* $\leftarrow$
    V_1 *: vector* $\wedge$ V_2 *: vector* $\wedge$
    α *is cos*$^{-1}$ $\frac{V_1.aV_2.a+V_1.bV_2.b+V_1.cV_2.c}{\sqrt{(V_1.a)^2+(V_1.b)^2+(V_1.c)^2}\sqrt{(V_2.a)^2+(V_2.b)^2+(V_2.c)^2}}$

When the angle between V$_1$ and V$_2$ equals -180 degrees then we don't need to do anything. This means that the relative orientation relation is already fullfilled. However, when this is not the case, we need to rotate the source concept over an angle of -180 - α (when *alpha* is the angle between V$_1$ and V$_2$). This rotation is done around the unit vector that stands perpendicular on the vectors V$_1$ and V$_2$. This perpendicular vector can be calculated as the cross product of V$_1$ and V$_2$.

The cross product $V_1 \times V_2$ is calculated as follows:

$$V_1 \times V_2 = (y_1z_2 - z_1y_2, z_1x_2 - x_1z_2, x_1y_2 - y_1x_2)$$

We call this vector V_3, $V_3 = V_1 \times V_2 = (x_3, y_3, z_3)$. We will now calculate the unit vector $u = (u_1, u_2, u_3)$ parallel to the vector V_3. The unit vector u is thus equal to $\frac{V_3}{||V_3||}$.

Now, in definition 7.6.5 the predicate *isOrthogonal(*V_1*,* V_2*, V)* is defined. This predicate is true when V is a unit vector that stands perpendicular on the vectors V$_1$ and V$_2$.

Definition 7.6.5

```
isOrthogonal($V_1$, $V_2$, V) $\leftarrow$
    $V_1$ : vector $\wedge$ $V_2$ : vector $\wedge$ V : vector $\wedge$
    $V_3$ [ a $\rightarrow$ $V_1$.b $V_2$.c - $V_1$.c $V_2$.b;
            b $\rightarrow$ $V_1$.c $V_2$.a - $V_1$.a $V_2$.c;
            c $\rightarrow$ $V_1$.a $V_2$.b - $V_1$.b $V_2$.a ] $\wedge$
    N is $\sqrt{V_3.a^2 + V_3.b^2 + V_3.c^2}$ $\wedge$
    V [ a $\rightarrow$ $\frac{V_3.a}{N}$; b $\rightarrow$ $\frac{V_3.b}{N}$; c $\rightarrow$ $\frac{V_3.c}{N}$]
```

So the source object needs to be rotated -180 - α degrees around the unit vector u. However, as we have seen earlier (see definition 6.2.1) the orientation of an object is expressedd using Euler angles. This means that an object is rotated β degrees around its front-to-back axis, σ degrees around its left-to-right axis and θ degrees around its top-to-bottom axis. Therefore we need to translate the rotatation of α degrees around an arbitrary axis parallel to the vector $u = (u_1, u_2, u_3)$ towards the corresponding Euler-angle representation. Therefore we can use the following formulas:

$$\beta = sin^{-1}(u_1u_2(1 - cos(\alpha)) + u_3sin(\alpha))$$
$$\sigma = tan^{-1}(\frac{u_1sin(\alpha) - u_2u_3(1 - cos(\alpha))}{1-(u_1^2+u_3^2)(1-cos(\alpha)}$$
$$\theta = tan^{-1}(\frac{u_2sin(\alpha) - u_1u_3(1 - cos(\alpha))}{1-(u_2^2+u_3^2)(1-cos(\alpha)}$$

In definition 7.6.6, the predicate *eulerAngles(β, σ, θ, V, α)* is defined. This predicate is true when β, σ and θ are the angles to rotate around the x-axis, y-axis and z-axis respectively so that these rotations have an equal result as when rotating the object α degrees around a unit vector V.

Definition 7.6.6

```
eulerAngles($\beta$, $\sigma$, $\theta$, V, $\alpha$) $\leftarrow$ V : vector $\wedge$
    $\beta$ is $sin^{-1}$(V.a * V.b * (1 - cos($\alpha$)) + V.c * sin($\alpha$)) $\wedge$
    $\sigma$ is $tan^{-1}$ (V.a * sin($\alpha$) - V.b * V.c * (1-cos($\alpha$))) /
        (1 - (V.$a^2$ + V.$c^2$) * (1-cos($\alpha$))) $\wedge$
    $\theta$ is $tan^{-1}$ (V.b * sin($\alpha$) - V.a * V.c * (1-cos($\alpha$))) /
        (1-(V.$b^2$ + V.$c^2$) * (1-cos($\alpha$))) $\wedge$
```

Using all of the above defined predicates we can now define the semantics of the relative orientation relation. The *relativeOrientation* predicate is defined in definition 7.6.7.

Definition 7.6.7

$$
\begin{aligned}
&\textit{relativeOrientation(A, C, E)} \leftarrow \textit{A : concept} \wedge \\
&\qquad \textit{C : concept} \wedge \\
&\qquad \textit{A[oriented@C} \bullet\!\!\rightarrow \textit{R]} \wedge \\
&\qquad \textit{directionVectors}(A, C, V_1, V_2) \wedge \\
&\qquad \textit{angleBetween}(V_1, V_2, \delta) \wedge \\
&\qquad \lambda \textit{ is } \text{-}180 \text{ - } \delta \wedge \\
&\qquad \textit{isOrthogonal}(V_1, V_2, V) \wedge \\
&\qquad \textit{eulerAngles}(\alpha, \beta, \gamma, V, \lambda) \\
&\qquad \textit{A[externalOrientation} \rightarrow E_A] \wedge \\
&\qquad \textit{E : orientation[frontAngle} \bullet\!\!\rightarrow E_A.\textit{frontAngle} + \alpha, \\
&\qquad\qquad\qquad \textit{leftAngle} \bullet\!\!\rightarrow E_A.\textit{leftAngle} + \beta, \\
&\qquad\qquad\qquad \textit{topAngle} \bullet\!\!\rightarrow E_A.\textit{topAngle} + \gamma]
\end{aligned}
$$

Now finally we can refine the *externalOrientation* method defined for the class *concept*. We already defined a first version of this method in definition 7.3.8. Now we extend this definition (see definition 7.6.8).

Definition 7.6.8

$$
\begin{aligned}
\textit{A[externalOrientation} \rightarrow \textit{E]} \leftarrow\ & \textit{A:concept} \wedge \textit{C: concept} \wedge \\
& \textit{E:orientation} \wedge \\
& \textit{(referencedExtOrientation(A, C, E)} \vee && (1) \\
& \textit{relativeOrientation(A, C, E))} && (2)
\end{aligned}
$$

So the value E of the *externalOrientation* method is the value for which either:

- E is the external orientation of C which is the reference part of A, hence E is the external orientation for A, or,
- A and C are related by means of a relative orientation relation and E is the external orientation for A so that the orientation relation with respect to C is respected.

In this section we have discussed the formal definitions of the relative orientation relations. In the next section we will look to the formalization of orientation by angle relations.

7.6.2 Orientation by angle relations

Next we need to define the formalization of the orientation by angle relation. Suppose now that we have a concept A which is oriented by means of an orientation by angle relation as graphically expressed in figure 7.5. This relation states that A is rotated 90 degrees over the top to bottom axis.

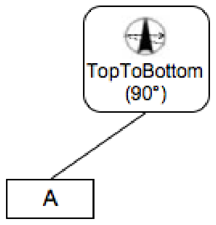


Figure 7.5: Example of an orientation by angle relation

More general, when a concept a is attached to an orientation by angle relation over the top to bottom axis with an angle of θ degrees, then this relation will be formalized by overwriting the externalOrient property as an inheritable property with value θ. The orientation by angle relation is formally defined in definition 7.6.9.

Definition 7.6.9 *Let a be a concept, thus a::concept. The fact that a has an orientation by angle relation around the front-to-back axis of α degrees, around the left-to-right axis of β degrees and around the top-to-bottom axis of θ degrees is expressed as follows:*

$$\begin{aligned} a[externalOrient \bullet\!\!\rightarrow orientation[&frontAngle \rightarrow \alpha, \\ &leftAngle \rightarrow \beta, \\ &topAngle \rightarrow \theta] \end{aligned}$$

So an orientation by angle relation is translated into its external orientation parameters.

Chapter 8

Formalizing Connection Relations

In this section we will explain how the connection point relation, the connection axis relation and the connection surface relation are formalized. Note that the formalization of the connection relations can be considered from two different viewpoints. On the one hand, a connection relation expresses how two components should be connected and in this way defines what this means in terms of the position and orientation of the two components. This is what we will call the *initial semantics* of the connection relation, as it defines the semantics of the connection relations at time zero of the virtual environment. On the other hand, a connection relation also expresses constraints on the possible behavior of the components during the rest of the lifetime of the connected objects. This is what we will call the *simulation semantics.*

8.1 Formalization of the connection axis relation

In this section we will formally define the connection axis relation. Because of the complexity of this connection relation, we will first elaborate on the mathematics underlying this relation. Afterwards, we will give the formal definition in F-Logic.

To establish a connection axis relation, five steps are needed. We will first describe each step from a mathematical point of view using an example.

1. **Compute the connection axes for both the source and the**

target objects.

As we have seen in the previous chapter, the connection axis for the

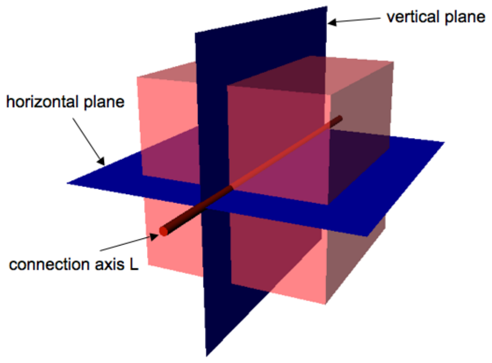


Figure 8.1: Connection axis is the intersection of two planes

source as well as for the target is described by means of two planes. The intersection of these planes forms the connection axis. In figure 8.1 a VR object is illustrated through which two planes (the horizontal and the vertical plane) are specified. We have to compute the intersection of the horizontal and the vertical plane. From mathematics we know that a line through the point (x_0, y_0, z_0) parallel with a vector (a, b, c) has the following parametric equations (for all t between -∞ and +∞):

$$\begin{cases} x = x_0 + ta \\ y = y_0 + tb \\ z = z_0 + tc \end{cases}$$

We know that the line which is the intersection of the horizontal and the vertical plane goes through the position of the object. So take the position of the object to be (x_0, y_0, z_0). We also know that the line is parallel to the vector $(1, 0, 0)$. Therefore we know that the parametric equations for the connection axis L (as the intersection of the horizontal plane and the vertical plane) will be as follows:

$$\begin{cases} x = x_0 + t \quad -\infty < t < +\infty \\ y = y_0 \\ z = z_0 \end{cases}$$

However it is possible that one, or both of the planes have been rotated around a certain axis. Take the example from figure 8.2. In this

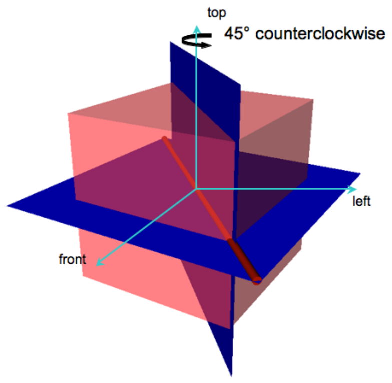


Figure 8.2: Rotation of the vertical plane around the top-to-bottom axis

example, the vertical plane has been rotated 45 degrees (or $\frac{\pi}{4}$ radians) counterclockwise around the top-to-bottom axis. From the definition of orientation (see 6.2.1) we know that a rotation around the top-to-bottom axis equals a rotation around the z-axis. Therefore we need to rotate the unit vector (1, 0, 0) (which is the intersection between the horizontal and vertical plane) also 45 degrees around the z-axis. This will result in a vector which is parallel to the line L which is the intersection of the horizontal plane and the rotated vertical plane. This rotation can be done using the following rotation matrix:

$$\begin{bmatrix} cos(\frac{\pi}{4}) & -sin(\frac{\pi}{4}) & 0 \\ sin(\frac{\pi}{4}) & cos(\frac{\pi}{4}) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

So the vector $v = (v_1, v_2, v_3)$ which is parallel to the intersection line between the horizontal and the rotated vertical plane is given by the multiplication of the above rotation matrix and the vector (1, 0, 0):

$$\begin{cases} v_1 = 1.cos(\frac{\pi}{4}) - 0.sin(\frac{\pi}{4}) = \frac{\sqrt{2}}{2} \\ v_2 = 1.sin(\frac{\pi}{4}) + y.cos(\frac{\pi}{4}) = \frac{\sqrt{2}}{2} \\ v_3 = 0 \end{cases}$$

So the line L, which is parallel to the vector $(\frac{\sqrt{2}}{2}, \frac{\sqrt{2}}{2}, 0)$ and goes trough the point (x_0, y_0, z_0) has the following equation:

$$\begin{cases} x = x_0 + \frac{\sqrt{2}}{2}t & -\infty < t < +\infty \\ y = y_0 + \frac{\sqrt{2}}{2}t \\ z = z_0 \end{cases}$$

The last possibility is that one of the planes has been translated in a certain direction. This will be reflected in the point (x_0, y_0, z_0). Suppose the vertical plane from our example in figure 8.1 has been translated 3 units to the left. Since the left direction corresponds to the y-axis, we know that the point (x_0, $y_0 + 3$, z_0) will lie on the intersection line between the horizontal plane and the translated vertical plane. Hence, this point (x_0, y_0+3, z_0) will be used in the parametric equations for the intersection line.
So far we have explained how to calculate the connection axis for the source and target objects of the connection axis relation.

2. **Rotate the source so that its connection axis is parallel to the connection axis for the target.**

   Suppose we have the situation as illustrated in figure 8.3. In this

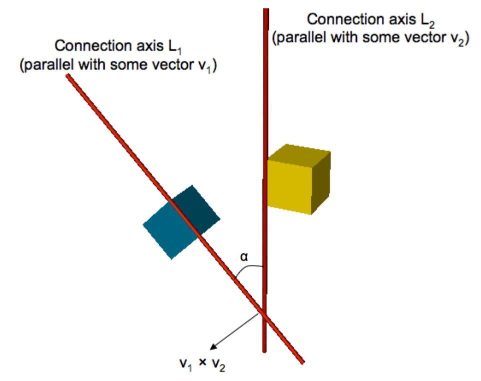

Connection axis L_1
(parallel with some vector v_1)
Connection axis L_2
(parallel with some vector v_2)
α
$v_1 \times v_2$

Figure 8.3: Connection axes for source and target may not fall together

situation the connection axes of the source and the target do not fall together. Since, for the connection axis relation it is required that both connection axes fall together we will need to adjust the situation. In this step we will change the orientation of the source (which is the blue cube in our example) so that its connection axis becomes parallel to the connection axis for the target (which is the yellow cube in the example).

First we will calculate the angle α between the connection axes L_1 and L_2. We know that the line L_1 is parallel with a vector $v_1 = (x_1, y_1, z_1)$ and that the line L_2 is parallel with a vector $v_2 = (x_2, y_2, z_2)$. As we have seen before with the formalization of the orientation relations, with this information we can calculate the angle α between both lines using the mathematical formulae:

$$\cos\alpha = \frac{v_1 \cdot v_2}{||v_1||||v_2||}$$

Since the dot product of v_1 and v_2 equals $x_1x_2 + y_1y_2 + z_1z_2$ and the norm of a vector $v = (x, y, z)$ equals $\sqrt{x^2 + y^2 + z^2}$ we know that we can calculate the angle α as:

$$\cos\alpha = \frac{x_1x_2+y_1y_2+z_1z_2}{\sqrt{x_1^2+y_1^2+z_1^2}\sqrt{x_2^2+y_2^2+z_2^2}}$$

When the angle α equals 0, then we don't need to change the orientation of the source object. An angle of 0 degrees means that the connection axes are already parallel. When the angle differs from 0, we need to calculate the vector $v_1 \times v_2$ which stands perpendicular on both L_1 and L_2. The cross product $v_1 \times v_2$ is calculated as follows:

$$v_1 \times v_2 = (y_1z_2 - z_1y_2, z_1x_2 - x_1z_2, x_1y_2 - y_1x_2)$$

We call this vector v_3, $v_3 = v_1 \times v_2 = (x_3, y_3, z_3)$. We will now calculate the unit vector $u = (u_1, u_2, u_3)$ parallel to the vector v_3. The unit vector u is thus equal to $\frac{v_3}{||v_3||}$. So the source object needs to be rotated α degrees around the unit vector u. However, as we have seen earlier (see definition 6.2.1) the orientation of an object is expressedd using Euler angles. This means that an object is rotated β degrees around its front-to-back axis, σ degrees around its left-to-right axis and θ degrees around its top-to-bottom axis. Therefore we need to translate the rotatation of α degrees around an arbitrary axis parallel to the vector $u = (u_1, u_2, u_3)$ towards the corresponding Euler-angle representation. This is very similar to what we have seen with the orientation relations. When adding the Euler angles to the external orientation of the source object, the source object will be oriented in

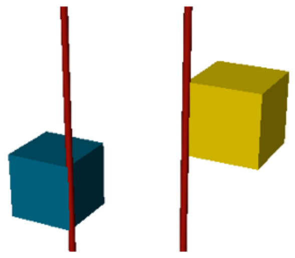


Figure 8.4: Connection axes for source and target are parallel

such a way that its connection axis is parallel to the connection axis of the target object. This situation is illustrated in figure 8.4.

3. **Calculate the orthogonal projections of the source and target position onto their connection axis.**

   The next things that we will calculate are the orthogonal projections of the position of the source and the target on their respective connection axes. These orthogonal projections are illustrated in figure 8.5 by means of p_s and p_t respectively. These projections will be the so called *translation points* for the source and target object. Suppose

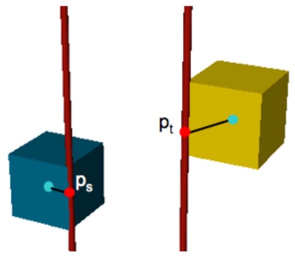


Figure 8.5: Orthogonal projection of source and target position on their connection axes

the source object is on position $p_0 = (x_0, y_0, z_0)$. We also know that the connection axis for the source, which we call L, goes through some point $p_1 = (x_1, y_1, z_1)$ and that it is parallel to some vector $v = (a, b, c)$.

This is all illustrated in figure 8.6.

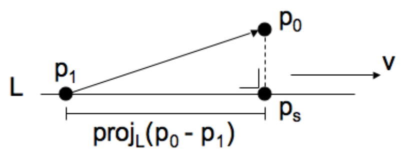


Figure 8.6: Orthogonal projection of a point on a line

To calculate the orthogonal projection p_s we first need to calculate the projection of $p_0 - p_1$ onto the line L. This is indicated in figure 8.6 as the line segment $proj_L(p_0 - p_1)$. To get the point p_s we need to add $proj_L(p_0 - p_1)\frac{v}{||v||}$ to p_1. So $p_s = p_1 + proj_L(p_0 - p_1)\frac{v}{||v||}$. The projection of $p_0 - p_1$ onto the line L can be calculated as follows:

$$proj_L(p_0 - p_1) = \frac{(p_0 - p_1) \cdot v}{||v||}$$

We can do this calculation for both the source and the target. So far we have calculated the translation points for the source object and the target object.

4. **Translate the source so that its connection axis falls together with the connection axis for the target.**
The next thing we need to do is to calculate the translation of the source object necessary to make sure that the connection axes and the translation points fall together. Therefore we will calculate the vector $v = p_t - ps$ as shown in figure 8.7. If we now translate the position of the source over the vector v, the source will be positioned in such a way that the connection axes fall together as do the translation points. The result in illustrated in figure 8.8

5. **Translate the source along the axis so that the translation point conditions of the connection axis relation are respected.**
The last step that may be necessary when solving a connection axis relation is meeting the translation point conditions. As we have seen in the informal description, the designer may specify a translation of one or both translation points along the connection axis. By doing this, the designer actually specifies the initial position of the connected objects somewhere along the connection axis. Now suppose in our case that

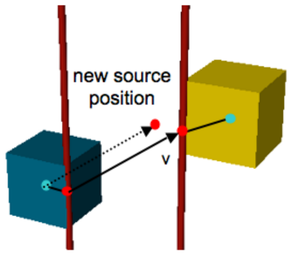


Figure 8.7: Calculation of the translation of the source object

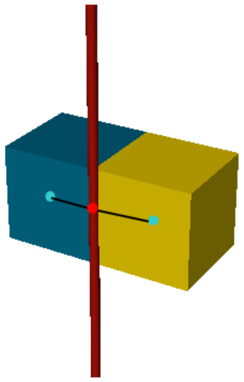


Figure 8.8: Result after the calculation of the connection axis relation

> the designer specified a translation of the source translation point 3 units towards the top of the connection axis. Take t to be the original translation point and t_n to be the translated translation point. Since we know the equation of the connection axis L, it is easy to calculate t_n. Next we can calculate the vector t_n - t. Now we can use this vector to translate the source object. This is illustrated in figure 8.9. The point p_s becomes the new position for the source object.

So far we have described the semantics of a connection axis relation from a mathematical point of view. Now we will formally define this complete relation description using F-Logic.

We will first define the horizontal plane, the vertical plane and the perpendicular plane. As we have seen earlier, the horizontal plane can be rotated

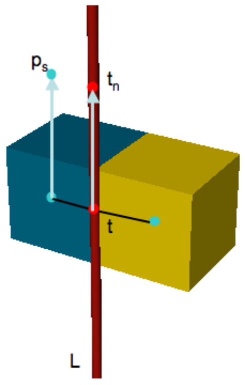


Figure 8.9: Translation of the translation point for the source object

over the left-to-right axis and over the front-to-back axis. The horizontal plane can also be translated along the top-to-bottom axis. This means that the point (0, 0, z) lies on the plane when the default horizontal plane has been translated z units along the top-to-bottom axis (z $\in \Re$).
So in our definition of the horizontal plane we have two properties which indicate the possible rotation around the left-to-right axis and the front-to-back axis (which have the default value 0). We also have three properties x_0, y_0 and z_0. x_0, y_0 and z_0 are defined as inheritable properties having the value 0 while the value of z_0 will then be overwritten with the value of the translation along the top-to-bottom axis. The formal definition of the horizontal plane is given in definition 8.1.1. Also note that we define the horizontal plane to be a subclass of plane. Note that plane is a class having no properties. We only included the class plane to be able to refer to a horizontal as well as to a vertical or a perpendicular plane.

Definition 8.1.1 *We formally define the horizontal plane as follows in F-logic:*

horizontal :: plane
horizontal [leftToRightAngle $\bullet\!\!\rightarrow$ *0;*
frontToBackAngle $\bullet\!\!\rightarrow$ *0;*
$x_0 \bullet\!\!\rightarrow 0;$
$y_0 \bullet\!\!\rightarrow 0;$
$z_0 \bullet\!\!\rightarrow 0\ ]$

We define the vertical and the perpendicular plane similar in respectively definition 8.1.2 and 8.1.3.

Definition 8.1.2 *We formally define the vertical plane as follows in F-logic:*

vertical :: plane
vertical [frontToBackAngle $\bullet\!\!\rightarrow$ *0;*
topToBottomAngle $\bullet\!\!\rightarrow$ *0;*
x_0 $\bullet\!\!\rightarrow$ *0;*
y_0 $\bullet\!\!\rightarrow$ *0;*
z_0 $\bullet\!\!\rightarrow$ *0;]*

Definition 8.1.3 *We formally define the perpendicular plane as follows in F-logic:*

perpendicular :: plane
perpendicular [leftToRightAngle $\bullet\!\!\rightarrow$ *0;*
topToBottomAngle $\bullet\!\!\rightarrow$ *0;*
x_0 $\bullet\!\!\rightarrow$ *0;*
y_0 $\bullet\!\!\rightarrow$ *0;*
z_0 $\bullet\!\!\rightarrow$ *0]*

Formalization of the initial semantics

First we will formalize the initial semantics of the connection axis relation. To do this we return to the five steps described above. For each of these steps we will formally define a number of predicates. Note that we will not explain these formal definitions in detail since they are a straightforward translation of the mathematical approach we have given.

1. **Compute the connection axes for both the source and the target objects.**

   First we will define a predicate *horizontalVerticalIntersect(P, Q, V)* which is true when a vector V is parallel to the intersection of an instance of horizontal plane and an instance of vertical plane. This predicate is formally defined in definition 8.1.4.

Definition 8.1.4

```
horizontalVerticalIntersect(P, Q, V) ←
    V : vector ∧ P : plane ∧ Q : plane ∧
    (    ( P : horizontal ∧ Q : vertical ∧
           P [ leftToRightAngle → α ] ∧
           Q [ topToBottomAngle → β ] ) ∨
         (P : vertical ∧ Q : horizontal ∧
          P [ topToBottomAngle → α ] ∧
          Q [ leftToRightAngle → β ] )) ∧
    V_1 : vector [ a → 1, b → 0, c → 0 ] ∧
    rotate(V_1, 0, α, β, V)
```

Note that we do not need to take into account rotations of the horizontal plane as well as of the vertical plane around the front-to-back axis. This is because the intersecting line falls together with this rotation axis and thus doesn't affect the orientation of the vector V parallel to the intersection line. Also remember that the predicate *rotate* which is used in the above definition was defined in definition 6.3.4.
It is easy to see that we can define similar predicates *horizontalPerpendicularIntersect(P, Q, V)* and *verticalPerpendicularIntersect(P, Q, V)* which are true when V is a vector parallel to the intersection of an instance of horizontal plane and an instance of perpendicular plane respectively parallel to the intersection of an instance of vertical plane with an instance of perpendicular plane.
Using these predicates we will now formally define the predicate *parallelToIntersection(P, Q, V)*. This predicate turns out to be true when a vector V is parallel to the intersection of two planes P and Q. This predicate is defined in definition 8.1.5.

Definition 8.1.5

```
parallelToIntersection(P, Q, V) ←
        V : vector ∧ P : plane ∧ Q : plane ∧
        ( horizontalVerticalIntersect(P, Q, V) ∨
          horizontalPerpendicularIntersect(P, Q, V) ∨
          verticalPerpendicularIntersect(P, Q, V))
```

Now finally we can define the predicate *isIntersectionLine(P, Q, L)* which is true when L is a line which is the intersection of a plane P and a plane Q. We define this predicate in definition 8.1.6.

Definition 8.1.6

```
isIntersectionLine(P, Q, L) ←
        L : line ∧ P : plane ∧ Q : plane ∧
        parallelToIntersection(P, Q, V) ∧
        L [ $x_0$ → P.$x_0$ + Q.$x_0$;
            $y_0$ → P.$y_0$ + Q.$y_0$;
            $z_0$ → P.$z_0$ + Q.$z_0$;
            a → V.a;
            b → V.b;
            c → V.c ]
```

Note the use of path expressions in definition 8.1.6. Now we have defined all necessary F-logic predicates for the first step.

2. **Rotate the source so that its connection axis is parallel to the connection axis for the target.**

   The first predicate that we will define in this second step is the predicate *angleBetween(L, M, α)* which is true when α is the angle between two lines L and M. This predicate is defined in definition 8.1.7. Note that this predicate is very similar to definition 7.6.4 we have seen in chapter 7. Actually, the predicate is overloaded so that it now works for angles between vectors as well as for angles between lines.

   Definition 8.1.7

```
angleBetween(L, M, α) ←
        L : line ∧ M : line ∧
        $\alpha \ is \ cos^{-1} \frac{L.aM.a+L.bM.b+L.cM.c}{\sqrt{(L.a)^2+(L.b)^2+(L.c)^2}\sqrt{(M.a)^2+(M.b)^2+(M.c)^2}}$
```

   Next we also need a predicate which is true when a unit vector V is orthogonal to two lines L and M. Therefore we define the predicate *isOrthogonalTo(L, M, V)*. This predicate is defined in definition 8.1.8. This predicate is also overloaded from definition 7.6.5.

   Definition 8.1.8

```
isOrthogonal(L, M, V) ←
      L : line ∧ M : line ∧ V : vector ∧
```

V_1 *: vector [* $a \rightarrow L.b\ M.c$ - $L.c\ M.b$,
$b \rightarrow L.c\ M.a$ - $L.a\ M.c$,
$c \rightarrow L.a\ M.b$ - $L.b\ M.a$ *]* $\wedge$
N *is* $\sqrt{V_1.a^2 + V_1.b^2 + V_1.c^2}$ $\wedge$
V *[* $a \rightarrow \frac{V_1.a}{N}$; $b \rightarrow \frac{V_1.b}{N}$; $c \rightarrow \frac{V_1.c}{N}$*]*

So far we have defined all the necessary predicates for the second step.

3. **Calculate the orthogonal projections of the source and target position onto their connection axis.**

   For this step we need to define a predicate which is true when a point P is the orthogonal projection of a point Q onto a line L. Therefore in definition 8.1.9 we formally define the predicate *projection(P, Q, L)*.

   Definition 8.1.9

   projection(P, Q, L) $\leftarrow$
   P : point $\wedge$ *Q : point* $\wedge$ *L : line* $\wedge$
   X_1 *is* $Q.x$ - $L.x_0$ $\wedge$ Y_1 *is* $Q.y$ - $L.y_0$ $\wedge$ Z_1 *is* $Q.z$ - $L.z_0$ $\wedge$
   N_V *is* $\sqrt{L.aL.a + L.bL.b + L.cL.c}$
   P_L *is* $\frac{X_1 * L.a + Y_1 * L.b + Z_1 * L.c}{N_V}$ $\wedge$
   P *[* $x \rightarrow L.x_0 + \frac{P_L * L.a}{N_V}$;
   $y \rightarrow L.y_0 + \frac{P_L * L.b}{N_V}$;
   $z \rightarrow L.z_0 + \frac{P_L * L.c}{N_V}$ *]*

4. **Translate the source so that its connection axis falls together with the connection axis for the target.**

   In this step we will formally define a predicate *isDifference(V,* V_1*,* V_2*)* which is true when a vector V is the result of the vector V_1 minus the vector V_2. This predicate is defined in definition 8.1.10.

   Definition 8.1.10

   isDifference(V, V_1*,* V_2*)* $\leftarrow$
   V : vector $\wedge$ V_1 *: vector* $\wedge$ V_2 *: vector* $\wedge$
   V *[* $a \rightarrow V_1.a$ - $V_2.a$;
   $b \rightarrow V_1.b$ - $V_2.b$;
   $c \rightarrow V_1.c$ - $V_2.c$; *]*

5. **Translate the source along the axis so that the translation point conditions of the connection axis relation are respected.**

   For this last step we will define two predicates. The first predicate is the *parameterValue(L, P, T)* predicate. This predicate is true when T is the value to fill in in the parametric equations of a line L to get the point P. This predicate is formally defined in definition 8.1.11.

   Definition 8.1.11

```
parameterValue(L, P, T) ←
        L : line ∧ P : point ∧
        $P.x = L.x_0 + T * L.a$ ∧
        $P.y = L.y_0 + T * L.b$ ∧
        $P.z = L.z_0 + T * L.c$
```

   Next in definition 8.1.12 we will define the predicate *newSourceTP(R, P, Q)*. Take P to be the translation point for the source object and R to be the connection axis relation involved. Note that the definition for the connection axis relation is given a bit further. The predicate *newSourceTP(R, P, Q)* is true when Q is the translation point for the source after translating the original source translation point P according to the connection axis relation R.

   Definition 8.1.12

```
newSourceTP(R, P, Q) ←
      R : connectionAxisRelation ∧ P : point ∧ Q : point ∧
      R [ sourceTPDist → D, sourceAxis → L ] ∧
      parameterValue(L, P, T) ∧
      ( ( R [ sourceDir → "left" ] ∧                              (1)
          ( ($L.b > 0$ ∧ S is T + D) ∨
            ($L.b < 0$ ∧ S is T-D))) ∨
        ( R [sourceDir → "top" ] ∧                                (2)
          ( ($L.c > 0$ ∧ S is T + D) ∨
            ($L.c < 0$ ∧ S is T-D))) ∨
        . . . ) ∧
      Q [ x → L.x + S * L.a;
          y → L.y + S * L.b;
          z → L.z + S * L.c ]
```

So for example, when the translation point needs to be translated into the left direction (see (1)) then based on the slope in the left direction (the slope in the y-direction) which is positive or negative we can calculate the new translation point along the line. Line (2) shows the case when the translation is in the top direction. It is easy to write similar approaches for possible translations in all other directions.
We have now defined all the necessary predicates for all five steps needed when solving a connection axis relation.

We will now use all of the above predicates to formally define the connection axis relation as well as the semantics of this relation. The connectionAxisRelation class defined in definition 8.1.13 contains a number of properties keeping track of the planes used to define the connection axis for both the source and target objects. There are also a number of properties which keep track of the translation for the translation point for the source or the target objects. Next the *connectionAxisRelation* class also has two properties *sourceAxis* and *targetAxis* which are methods returning a line which is the connection axis for the source and the target object respectively.

Definition 8.1.13 *We formally define the connection axis relation as follows in F-logic:*

$$\begin{aligned}
&\textit{connectionAxisRelation}\ [\ \textit{sourcePlane1} \Rightarrow \textit{plane};\\
&\qquad \textit{sourcePlane2} \Rightarrow \textit{plane};\\
&\qquad \textit{targetPlane1} \Rightarrow \textit{plane};\\
&\qquad \textit{targetPlane2} \Rightarrow \textit{plane};\\
&\qquad \textit{sourceAxis} \Rightarrow \textit{line};\\
&\qquad \textit{targetAxis} \Rightarrow \textit{line};\\
&\qquad \textit{sourceTPDist} \Rightarrow \textit{float};\\
&\qquad \textit{sourceTPDir} \Rightarrow \textit{string};\\
&\qquad \textit{targetTPDist} \Rightarrow \textit{float};\\
&\qquad \textit{targetTPDir} \Rightarrow \textit{string}\ ]
\end{aligned}$$

The methods sourceAxis and targetAxis are defined as follows:

$$\begin{aligned}
C[\textit{sourceAxis} \rightarrow L] \leftarrow\ & C : \textit{connectionAxisRelation} \wedge L : \textit{line} \wedge\\
& A[\textit{connectedTo}@B \bullet\!\!\rightarrow C] \wedge\\
& A[\textit{position} \rightarrow p] \wedge\\
& \textit{intersectionLine}(C.\textit{sourcePlane1},\ C.\textit{sourcePlane2},\ M) \wedge\\
& L : \textit{line}[x_0 \rightarrow M.x_0\ +\ p.x;\\
& \qquad\quad y_0 \rightarrow M.y_0\ +\ p.y;
\end{aligned}$$

$z_0 \rightarrow M.z_0 + p.z;$
$a \rightarrow M.a;\ b \rightarrow M.b;\ c \rightarrow M.c\ ]$

$C[targetAxis \rightarrow L] \leftarrow C : connectionAxisRelation \wedge L : line \wedge$
$A[connectedTo@B \bullet\!\!\rightarrow C] \wedge$
$B[position \rightarrow p] \wedge$
$intersectionLine(C.targetPlane1,\ C.targetPlane2,\ M) \wedge$
$L : line[x_0 \rightarrow M.x_0 + p.x;$
$y_0 \rightarrow M.y_0 + p.y;$
$z_0 \rightarrow M.z_0 + p.z;$
$a \rightarrow M.a;\ b \rightarrow M.b;\ c \rightarrow M.c\ ]$

We will now show how a connection axis relation between two concepts is formally defined. This is done in definition 8.1.14.

Definition 8.1.14 *Let a and b be two concepts, thus a :: concept and b :: concept. The fact that a is connected to b by means of a connection axis relation r (thus r : connectionAxisRelation) is formally expressed as follows:*

$a\ [\ connectedTo@b \bullet\!\!\rightarrow r\ ]$

Next we will formally define the initial semantics of a connection axis relation between two concepts in terms of the position and orientation of the source concept involved in the relation.

First we will define the predicate *connectionAxisPos(A, C, P)* which is true when P is the position of a concept A when it is connected via a connection axis relation to concept C. This predicate is defined in definition 8.1.15. Actually, definition 8.1.15 contains the complete process we discussed in the five steps above. It unites all the predicates defined above.

Definition 8.1.15
$connectionAxisPos(A,\ C,\ P) \leftarrow A : concept \wedge C : concept \wedge P : point \wedge$
$A\ [\ connectedTo@C \bullet\!\!\rightarrow R] \wedge$
$R\ [sourceAxis \rightarrow L,\ targetAxis \rightarrow M\ ] \wedge$
$projection(T_1,\ A.position,\ L) \wedge$
$projection(T_2,\ C.position,\ M) \wedge$
$V_1 : vector[\ a \rightarrow T_1.x,\ b \rightarrow T_1.y,\ c \rightarrow T_1.z] \wedge$
$V_2 : vector[\ a \rightarrow T_2.x,\ b \rightarrow T_2.y,\ c \rightarrow T_2.z] \wedge$
$isDifference(V,\ V_2,\ V_1) \wedge$

P_1 *: point[* $x \rightarrow$ *A.position.x + V.a,*
$y \rightarrow$ *A.position.y + V.b,*
$z \rightarrow$ *A.position.z + V.c]* $\wedge$
projection(T_3, P_1, *M)* $\wedge$
newSourceTP(R, T_3, T_4*)* $\wedge$
isDifference(T, T_4, T_3*)* $\wedge$
P : point[$x \rightarrow P_1.x + T.a,$
$y \rightarrow P_1.y + T.b,$
$z \rightarrow P_1.z + T.c$ *]*

So the above predicate can check the position of a concept A (the source) which is connected to a concept C (the target) by means of a connection axis relation. However, not only the position of A is stipulated by the connection axis relation, and also the position of the target concept C can be influenced by this connection relation. This is because there may be specified a translation of the target translation point. Therefore we define a predicate *TPPos(C, R, P)* which is true when P is the position of of a concept C for C playing the role of target object in some connection axis relation R. This predicate is defined in definition 8.1.16. Note the predicate *newTargetTP* is defined similar to the predicate *newSourceTP* (see definition 8.1.12).

Definition 8.1.16
TPPos(C, R, P) $\leftarrow$ *C : concept* $\wedge$ *R : connectionAxisRelation* $\wedge$ *P : point* $\wedge$
R [targetAxis $\rightarrow$ *L]* $\wedge$
projection(T_1*, C.position, L)* $\wedge$
newTargetTP(R, T_1, T_2*)* $\wedge$
isDifference (T, T_2, T_1*)* $\wedge$
P [$x \rightarrow$ *C.position.x + T.a,*
$y \rightarrow$ *C.position.y + T.b,*
$z \rightarrow$ *C.position.z + T.c]*

Next we will define a predicate *connectedPosition(A, C, P)* which is true when P is the position of a concept A which is connected via a connection relation (not necessarily a connection axis relation) to a concept C. The predicate works as well when A is the source of a connection relation as when A is the target. Note that in the first definition we will only take into account connection axis relations. Later when we formalize the connection point relation and the connection surface relation we will adapt this definition. The predicate is formally defined in definition 8.1.17.

Definition 8.1.17

connectedPosition(A, C, P) $\leftarrow$ *A : concept* $\wedge$ *C : concept* $\wedge$ *P : point* $\wedge$
  (A [connectedTo@C $\bullet\!\!\rightarrow$ *R]* $\wedge$ *R : connectionAxisRelation* $\wedge$
  connectionAxisPos(A, C, P)) $\vee$
  (C [connectedTo@A $\bullet\!\!\rightarrow$ *R]* $\wedge$ *R : connectionAxisRelation* $\wedge$
  TPPos(A, R, P))

We now only need to extend the method *position* which was defined for *concept* earlier in definition 7.5.10. Earlier we defined the position method returning the position of a concept when (1) the concept was a complex concept having a reference concept, or when (2) the concept was related by means of a spatial relation to another concept. Now we extend this signature so that the method also returns the relative position of the concept when it is connected to another concept by means of a connection relation. The extended definition is given in definition 8.1.18.

Definition 8.1.18

A[position $\rightarrow$ *P]* $\leftarrow$ *A:concept* $\wedge$ *C:concept* $\wedge$ *P:point* $\wedge$
  (referencedPosition(A, C, P) $\vee$ *(1)*
  spatialPosition(A, C, P) $\vee$ *(2)*
  connectedPosition(A, C, P)) *(3)*

So far we have defined the semantics of the connection axis relation in terms of positioning. However it is also possible that the external orientation of the source object changes when it is connected to another concept, as we have seen in step 2. Therefore we will define the predicate *connectionAxisOrient(A, C, E)* which is true when E is the external orientation a concept A needs to have so that it's connection axis becomes parallel to the connection axis of a concept C it is connected to. This predicate is defined in definition 8.1.19.

Definition 8.1.19

connectionAxisOrient(A, C, E) $\leftarrow$
  A : concept $\wedge$ *C : concept* $\wedge$ *E : orientation* $\wedge$
  A [connectedTo@C $\bullet\!\!\rightarrow$ *R]* $\wedge$
  R : connectionAxisRelation $\wedge$
  R [sourceAxis $\rightarrow$ *L, targetAxis* $\rightarrow$ *M]* $\wedge$
  angleBetween(L, M, α*)* $\wedge$
  isOrthogonal(L, M, V) $\wedge$
  eulerAngles(β*,* σ*,* θ*, V,* α*)* $\wedge$

A[externalOrientation $\rightarrow$ E_A*]* $\wedge$
E : orientation[frontAngle $\rightarrow \beta$ + E_A*.frontAngle;*
leftAngle $\rightarrow \sigma$ + E_A*.leftAngle;*
topAngle $\rightarrow \theta$ + E_A*.topAngle]*

We also define a predicate *connectedOrientation(A, C, E)* which is true when E is the external orientation for a concept A which respects the connection relation requirements when A is connected to C by means of some connection relation. This predicate is defined in definition 8.1.20.

Definition 8.1.20
connectedOrientation(A, C, E) $\leftarrow$
A : concept $\wedge$ *C : concept* $\wedge$ *E : orientation* $\wedge$
A [connectedTo@C $\rightarrow$ *R]* $\wedge$
R : connectionAxisRelation $\wedge$ *connectionAxisOrient(A, C, E)*

Now we also need to adapt the method *externalOrientation* defined for concepts (see definition 7.6.8). We will extend this definition so that the external orientation method can also return the external orientation of a concept A connected to a concept C. The extended definition is given in definition 8.1.21.

Definition 8.1.21
A[externalOrientation $\rightarrow$ *E]* $\leftarrow$ *A:concept* $\wedge$ *C: concept* $\wedge$
E:orientation $\wedge$
(referencedExtOrientation(A, C, E) $\vee$
relativeOrientation(A, C, E) $\vee$
connectedOrientation(A, C, E))

Formalization of the simulation semantics

So far we have formalized the initial semantics of the connection axis relation. However, we also need to define the meaning of the connection axis relation for the rest of the simulation, the so called *in simulation semantics.* In the initial semantics, the source needs to be positioned and oriented according to the targets position and orientation taking into account the connection axis relation. During the simulation, the difference between source and target is not relevant anymore. When one of the objects moves or changes orientation, the other one has to move with it in such a way that their connection axis relation is respected.

First we define the predicate *caPosConstraint(A, C, P)* which is true when P is the position of A relative to C according to a connection axis relation between A and C. Note that this definition is very similar to the definition of the predicate *connectionAxisPos* (see definition 8.1.15). However, in the predicate *connectionAxisPos* we positioned the source along the connection axis according to the specified translation point. Now for the simulation semantics, the translation points are of zero importance. Therefore instead of using the distance between both translation points (see step 4) to move the source towards the target, we will use the distance between both connection axes.

Therefore we first need to define a predicate $lineLineDist(L, M, V)$. As we assume lines L and M to be parallel (this is the case because of the enforced orientation by the connection axis relation), the *lineLineDist(L, M, V)* predicate is true when V is a vector from line L perpendicular to line M. The *lineLineDist* predicate is defined in definition 8.1.22.

Definition 8.1.22
lineLineDist(L, M, V) $\leftarrow$
    L : line $\wedge$ *M.line* $\wedge$
    P : point [$x \rightarrow L.x_0$; $y \rightarrow L.y_0$; $z \rightarrow L.z_0$*]* $\wedge$
    projection(P, Q, M) $\wedge$
    V_1 *: vector [* $a \rightarrow L.x_0$; $b \rightarrow L.y_0$; $c \rightarrow L.z_0$*]* $\wedge$
    V_2 *: vector [* $a \rightarrow Q.a$; $b \rightarrow Q.b$; $c \rightarrow Q.c$*]* $\wedge$
    isDifference(V, V_2, V_1*)*

Now we can use the *lineLineDist* predicate to define the *caPosConstraint(A, C, P)* predicate. This is given in definition 8.1.23.

Definition 8.1.23
caPosConstraint(A, C, P) $\leftarrow$ *A : concept* $\wedge$ *C : concept* $\wedge$ *P : point* $\wedge$
    ((A [connectedTo@C $\rightarrow$ *R]* $\wedge$
        R [sourceAxis $\rightarrow$ *L, targetAxis* $\rightarrow$ *M])* $\vee$
      (C [connectedTo@A $\rightarrow$ *R]* $\wedge$
        R [targetAxis $\rightarrow$ *L, sourceAxis* $\rightarrow$ *M]))* $\wedge$
    A_S *: line [* $x_0 \rightarrow A.position.x + L.x_0$;
                $y_0 \rightarrow A.position.y + L.y_0$;
                $z_0 \rightarrow A.position.z + L.z_0$;
                $a \rightarrow L.a$; $b \rightarrow L.b$; $c \rightarrow L.c$ *]* $\wedge$
    A_C *: line [* $x_0 \rightarrow C.position.x + M.x_0$;

$y_0 \rightarrow$ *C.position.y + M.*y_0*;*
$z_0 \rightarrow$ *C.position.z + M.*z_0*;*
$a \rightarrow$ *M.a; b* $\rightarrow$ *M.b; c* $\rightarrow$ *M.c]* $\wedge$
lineLineDist(A_S*,* A_C*, V]* $\wedge$
P : point[x $\rightarrow$ *A.position.x + V.a,*
y $\rightarrow$ *A.position.y + V.b,*
z $\rightarrow$ *A.position.z + V.c]*

Now using the above predicate we can define the simulation semantics for the connection axis relation. This is formally defined in definition 8.1.24.

Definition 8.1.24 *If two concepts a and b (thus a :: concept and b :: concept) are connected over a connection axis relation R, then for their simulation semantics the position and orientation of a and b must be so that:*

caPosConstraint(a, b, P_a*)* $\wedge$ *a[position* $\rightarrow$ P_a*]*
caPosConstraint(b, a, P_b*)* $\wedge$ *b[position* $\rightarrow$ P_b*]*

connectionAxisOrient(a, b, E_a*)* $\wedge$ *a[externalOrientation* $\rightarrow$ E_a*]*
connectionAxisOrient(b, a, E_b*)* $\wedge$ *a[externalOrientation* $\rightarrow$ E_b*]*

We have now described the complete formalization of the connection axis relation. In the next section we will formally define the connection point relation.

8.2 Formalization of the connection point relation

In this section we will formally define the connection point relation. We informally introduced this relation in chapter 4. Remember that on both objects which are connected by means of a connection point relation a connection point is specified. The connection points of these objects need to be at the same position during the complete lifetime of the connected objects. We will start by the definition of the connection point relation. The connection point relation is defined in definition 8.2.1.

Definition 8.2.1

connectionPointRelation[sourceDirection $\Rightarrow$ *string;*
sourceDistance $\Rightarrow$ *float;*
targetDirection $\Rightarrow$ *string;*
targetDistance $\Rightarrow$ *float]*

So a connection point relation has four properties. The *sourceDirection* and *sourceDistance* properties together specify the connection point of the source object relative to the position point of the source. The *targetDirection* and *targetDistance* properties specify the connection point of the target object relative to the position of the target. Now we will define how to specify that two concepts are connected by means of some connection point relation. This is defined in definition 8.2.2.

Definition 8.2.2 *Let a and b be two concepts, thus a :: concept and b :: concept. The fact that a is connected to b by means of a connection point relation r (thus r : connectionPointRelation) is formally expressed as follows:*

a [connectedTo@b $\bullet\!\!\rightarrow$ *r]*

In the following paragraphs we will formalize the semantics of the connection point relation. Similar to the connection axis relation we also define two types of semantics. First we will formalize the initial semantics. Next we will define the simulation semantics of the connection point relation.

Formalization of the initial semantics

We will first define the predicate *targetCPPosition(T, R, P)* which is true when P is the connection point for a given target concept T according to a connection point relation R. This predicate is defined in defintion 8.2.3.

Definition 8.2.3
targetCPPosition(T, R, P) $\leftarrow$
  T : concept $\wedge$ *R : connectionPointRelation* $\wedge$
  R [targetDistance $\rightarrow$ *D]* $\wedge$
  ((R [targetDirection $\rightarrow$ *"left"]* $\wedge$ *relativePosition(T, 0, D, 0, P))* $\vee$
    (R [targetDirection $\rightarrow$ *"right"]* $\wedge$ *relativePosition(T, 0, -D, 0, P)* $\vee$
    (R [targetDirection $\rightarrow$ *"top"]* $\wedge$ *relativePosition(T, 0, 0, D, P)* $\vee$
    . . .)

Now we will first overload the predicate *relativePosition(A, X, Y, Z, P)* (see definition 7.5.3). The overloaded predicate will take one extra argument. The new predicate is defined in definition 8.2.4. This predicate is true when P is a point which lies X units in front of, Y units left of and Z units towards the top of some point Q, according to the orientation of some concept A.

Definition 8.2.4
relativePosition(A, Q, X, Y, Z, P) $\leftarrow$
  A : concept $\wedge$ *Q : point* $\wedge$
  iFrontAngle(A, iα) $\wedge$ *iLeftAngle(A, iβ)* $\wedge$
  iTopAngle(A, iσ) $\wedge$ *eFrontAngle(A, eα)* $\wedge$
  eLeftAngle(A, eβ) $\wedge$ *eTopAngle(A, eσ)* $\wedge$
  P_1 *: point[x* $\rightarrow$ *X, y* $\rightarrow$ *Y, z* $\rightarrow$ *Z]* $\wedge$
  rotate(P_1, iα, iβ, iσ, P_2) $\wedge$
  rotate(P_2, eα, eβ, eσ, P_3) $\wedge$
  P : point[x $\rightarrow$ P_3*.x + Q.x, y* $\rightarrow$ P_3*.y + Q.y, z* $\rightarrow$ P_3*.z + Q.z]*

Next we define the predicate *sourcePositionCP(S, T, R, P)*. This predicate is true when P is the position of the source object S which is connected via a connection point relation R to the target object T. This predicate is defined in definition 8.2.5.

Definition 8.2.5
sourcePositionCP(S, T, R, P) $\leftarrow$
  S : concept $\wedge$ *T : concept* $\wedge$ *R : connectionPointRelation* $\wedge$
  R [sourceDistance $\rightarrow$ *D]* $\wedge$
  targetCPPosition(T, R, P_1) $\wedge$
  ((R [sourceDirection $\rightarrow$ *"left"]* $\wedge$ *relativePosition(S, P_1, 0, -D, 0, P))* $\vee$
    (R [sourceDirection $\rightarrow$ *"right"]* $\wedge$ *relativePosition(S, P_1, 0, D, 0, P)* $\vee$
    (R [sourceDirection $\rightarrow$ *"top"]* $\wedge$ *relativePosition(S, P_1, 0, 0, -D, P)* $\vee$
    . . .)

So far we have defined some predicates to check the position of the source concept according to some connection point relation it is involved in. Now we will extend the definition of the *connectedPosition(A, C, P)* predicate which we defined in definition 8.1.17. We will extend this definition so that the predicate is true when P is the position of a concept A which is connected via a connection point relation or via a connection axis relation to some concept C.

Definition 8.2.6
connectedPosition(A, C, P) $\leftarrow$ *A : concept* $\wedge$ *C : concept* $\wedge$ *P : point* $\wedge$
(*A [connectedTo@C* $\bullet\!\!\rightarrow$ *R]* $\wedge$ *R : connectionPointRelation* $\wedge$
sourcePositionCP(A, C, R, P)) $\vee$
(*A [connectedTo@C* $\bullet\!\!\rightarrow$ *R]* $\wedge$ *R : connectionAxisRelation* $\wedge$
caPos(A, C, P)) $\vee$
(*C [connectedTo@A* $\bullet\!\!\rightarrow$ *R]* $\wedge$ *R : connectionAxisRelation* $\wedge$
TPPos(A, R, P))

Note that by extending the *connectedPosition* predicate the *position* method defined for concepts is now possible of returning the position of a concept which is connected to another concept by means of a connection point relation.

Formalization of the simulation semantics

So far we have formally defined the initial semantics of the connection point relation. However, during the rest of the lifetime of the connected objects, when one object moves, the other object may also need to change its position so that the connection point relation is respected. Actually there is no difference between source and target anymore. Remember that we only needed a distinction between source and target to know which object initially should be connected to the other object. We need to define an extra constraint so that when the source changes its position, the target eventually moves with it and vice versa. Therefore we define the predicate *cpPosConstraint(A, C, P)* which is true when P is the position of some concept A so that A is positioned in such a way that its connection point falls together with the connection point of some concept C according to some connection point relation R between A and C. This predicate is defined in definition 8.2.7.

Definition 8.2.7

```
cpPosConstraint(A, C, P) ←
    A : concept ∧ C : concept ∧ P : point ∧
    ( ( A [ connectedTo@C •→ R] ∧ sourcePositionCP(A, C, R, P)) ∨
      ( C[connectedTo@A •→ R] ∧
        R_1 : connectionPointRelation[sourceDistance → R.targetDistance;
                                      targetDistance → R.sourceDistance;
                                      sourceDirection → R.targetDirection;
                                      targetDirection → R.sourceDirection] ∧
        sourcePositionCP(A, C, R_1, P)))
```

So when A is the source for the connection point relation we can reuse the *sourcePositionCP* predicate to check its position. When A is the target then we create a new instance of the connection point relation. This new instance is actually the inverse of the original connection point relation between A and C. This means that the sourceDirection becomes the targetDirection and so on. This way we again can reuse the *sourcePositionCP* predicate but then with the new created connection point relation. So this predicate has to be true during the complete lifetime of two concepts connected via a connection point relation. This is defined in definition 8.2.8.

Definition 8.2.8 *Take a and b to be two concepts, thus a :: concept and b :: concept. When a[connectedTo@b → R] holds for some connection point relation R (thus R : connectionpointRelation) then the position of a and b during the complete lifetime of a and b in the virtual environment needs to be as follows:*

cpPosConstraint(a, b, P_a) ∧ a[position → P_a]
cpPosConstraint(b, a, P_b) ∧ b[position → P_b]

Now we have defined the initial semantics as well as the simulation semantics for the connection point relation. In the next section we will formalize the connection surface constraint.

8.3 Formalization of the connection surface relation

In this section we will formally define the connection surface relation. We will start with the definition of the connection surface relation.

To specify a connection surface relation between two objects, the designer has to specify a plane on both objects. These planes have to fall together in the virtual world at all times. For the formal definition we can use one of the plane subclasses *horizontal*, *vertical* or *perpendicular* as defined for the connection axis relation. Next the designer can also specify a translation point for both objects to define the initial position of the objects along the plane. A translation point can be translated into two directions. Therefore a translation point specification is defined as in definition 8.3.1. So a translation point specification for the connection surface relation has four properties, two directions indicating the directions of the translation and two distances indicating the distances for the translation.

Definition 8.3.1
csTranslationPoint [direction1 ⇒ string;
distance1 ⇒ float;
direction2 ⇒ string;
distance2 ⇒ float]

Using the plane subclasses and the *csTranslationPoint* class we can define the connection surface relation as in definition 8.3.2.

Definition 8.3.2
connectionSurfaceRelation [sourceSurface ⇒ plane;
targetSurface ⇒ plane;
sourceTP ⇒ csTranslationPoint;
targetTP ⇒ csTranslationPoint]

In definition 8.3.3 we will define how to specify that two concepts are connected by means of some connection surface relation.

Definition 8.3.3 *Let a and b be two concepts, thus a :: concept and b :: concept. The fact that a is connected to b by means of a connection surface relation r (thus r : connectionSurfaceRelation) is formally expressed as follows:*

a [connectedTo@b $\bullet\!\!\rightarrow$ *r]*

Now we can formally define the initial and simulation semantics for the connection surface relation.

Formalization of the initial semantics

The initial semantics for the connection surface relation is very similar to the initial semantics of the connection axis relation. We also need to take five steps. These steps are described below.

1. **Calculate the connection surface on both the source and target object.**

   The connection surface of one of the connected objects is specified by translating and/or rotating one of the default planes (horizontal, perpendicular or vertical plane). The calculation of the specified plane is included in the definition of the default planes we defined earlier.

2. **Rotate the source object so that the connection surface of the source objects becomes parallel to the connection surface of the target object.**

   Suppose we have the situation as shown in figure 8.10 where the

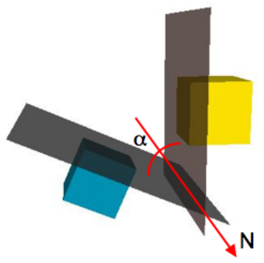


Figure 8.10: Rotation of the connection planes

connection surfaces of the connected objects are not parallel. Similar as we have seen for the connection axis relation, we will rotate the source object α degrees around the vector N which is parallel to the intersecting line between two connection surfaces. So therefore we

need to calculate the angle between the connection surfaces.
The angle between two planes (or surfaces) is equal to 180 - the angle between the normal vectors of the planes. Suppose the connection surface for the source object has the normal vector n_1 and the connection surface for the target object has the normal vector n_2, then $\cos \alpha = n_1 \cdot n_2$. Therefore, the angle between the planes equals 180 - α. The vector around which we need to rotate will be parallel to the vector which stand orthogonal on both $n_1$and n_2. Thus the vector around which we need to rotate is $n_1 \times n_2$.

3. **Calculate the orthogonal projections of the position of the objects onto their respective connection surfaces.**

   Similar to the connection axis relation where we calculated the orthogonal projection of the source and target position onto their respective connection axis to get the translation points, also here we need to calculate the translation points. However, now we need the orthogonal projections of the source and the target objects onto their respective connection surfaces.

4. **Translate the source so that the connection surface of the source falls together with the connection surface of the target.**

   So far we know that the connection surfaces are parallel to each other. We also know the initial translation points of both objects to connect. Next, we need to make sure that the connection surfaces fall together. Similar to the connection axis relation, we will calculate the vector between the translation points. Then we will translate the source object along this vector. This way, the connected objects are positioned in such a way that both their connection surfaces and their translation points fall together.

5. **Translate the source and the target inside the plane to respect the translation point requirements.**

   In the last step we need to calculate the translation of the source and target according to the specification of the translation points. Also this is very similar to the connection axis relation. The only difference is that now the translation point of a connected object can be translated in two directions instead of one direction as in the case of the connection axis relation.

So far we have given the mathematical description of the initial semantics for the connection surface relation. We will now give the F-logic formalization for this relation. Note that we can reuse most of the predicates defined for the connection axis relation.

For the first and second step we don't need to define additional predicates. These steps can be performed by predicates already defined for the connection axis relation. However, for the third step we need to define some new predicates. First we will define the predicate *planeParameters(S, N, P)* which is true when N is the normal vector for a given plane S (which can be an instance of *horizontal*, *perpendicular* or *vertical*) and P is a point on that plane. This predicate is defined in definition 8.3.4.

Definition 8.3.4

```
$planeParameters(S,\ N,\ P) \leftarrow S : plane \wedge$
    $(\ (\ S : horizontal \wedge$
        $S\ [\ frontToBackAngle \rightarrow \alpha;$
            $leftToRightAngle \rightarrow \beta] \wedge$
        $V : vector\ [a \rightarrow 0;\ b \rightarrow 0;\ c \rightarrow 1] \wedge$
        $rotate(V,\ \alpha,\ \beta,\ 0,\ N)\ ) \vee$
      $(\ S : vertical \wedge$
        $S\ [\ frontToBackAngle \rightarrow \alpha;$
            $topToBottomAngle \rightarrow \beta] \wedge$
        $V : vector\ [a \rightarrow 0;\ b \rightarrow 1;\ c \rightarrow 0] \wedge$
        $rotate(V,\ \alpha,\ 0,\ \beta,\ N)\ ) \vee$
      $(\ S : perpendicular \wedge$
        $S\ [\ leftToRightAngle \rightarrow \alpha;$
            $topToBottomAngle \rightarrow \beta] \wedge$
        $V : vector\ [a \rightarrow 1;\ b \rightarrow 0;\ c \rightarrow 0] \wedge$
        $rotate(V,\ 0,\ \alpha,\ \beta,\ N)\ )\ ) \wedge$
    $P : point\ [\ x \rightarrow S.x_0;\ y \rightarrow S.y_0;\ z \rightarrow S.z_0]$
```

Next we will define the predicate *planeProjection(P, S, Q)* which is true when a point P is the orthogonal projection of a point Q on a plane S. This predicate is defined in definition 8.3.5.

Definition 8.3.5

$planeProjection(P,\ S,\ Q) \leftarrow S : plane \wedge Q : point \wedge$

```
    planeParameters(S, N, $P_0$) $\wedge$
    L : line [ $x_0 \rightarrow P_0.x$; $y_0 \rightarrow P_0.y$; $z_0 \rightarrow P_0.z$;
            $a \rightarrow N.a$; $b \rightarrow N.b$; $c \rightarrow N.c$;] $\wedge$
    projection($P_1$, Q, L) $\wedge$
    V : vector[$a \rightarrow P_1.x$ - $P_0.x$; $b \rightarrow P_1.y$ - $P_0.y$; $c \rightarrow P_1.z$ - $P_0.z$] $\wedge$
    P : point [ $x \rightarrow Q.x$ - V.a; $y \rightarrow Q.y$ - V.b; $z \rightarrow Q.z$ - V.c]
```

So far we have defined the necessary predicates for step 3. For the 4th step we also can reuse existing predicates defined for the connection axis relation. For step 5 we need to define a number of predicates defining the translation point semantics for translation points on connection surfaces. The main difference from translation point on a connection axis is that for connection surfaces the translation point may be translated in two directions while for a connection axis it may be translated in only one direction.

Take for example the horizontal plane. This plane lies in the front-back and left-right directions while the top-bottom direction is the normal vector for the horizontal plane. Now suppose the horizontal plane has been rotated. This means that the original normal vector (which is the vector $(0, 0, 1)$) has been rotated in the same way as the plane is rotated. We will now introduce a predicate *planeRF(S, V_x, V_y, V_z)* which is true when the vectors V_x, V_y and V_z are the resulting vectors from rotating the vectors (1, 0, 0), (0, 1, 0) and (0, 0, 1) respectively in the same way as the plane S has been rotated. With other words, V_x indicates the front-to-back direction along the plane S, V_y the left-to-right direction along the plane and V_z the top-to-bottom direction along the plane. Note that for each plane subclass, one of these vectors will be the normal vector. This predicate is defined in definition 8.3.6.

Definition 8.3.6

```
planeRF(S, $V_x$, $V_y$, $V_z$) $\leftarrow$
    ( ( S : horzontal $\wedge$
        S [frontToBackAngle $\rightarrow \alpha$;
           leftToRightAngle $\rightarrow \beta$ ] $\wedge$
        $\theta$ is 0 ) $\vee$
      ( S : vertical $\wedge$
        S [frontToBackAngle $\rightarrow \alpha$;
           topToBottomAngle $\rightarrow \theta$ ] $\wedge$
        $\beta$ is 0 ) $\vee$
      ( S : perpendicular $\wedge$
            S [leftToRightAngle → β;
                topToBottomAngle → θ ] ∧
            α is 0 )) ∧
        X : point [x → 1; y → 0; z → 0 ] ∧
        Y : point [x → 0; y → 1; z → 0 ] ∧
        Z : point [x → 0; y → 0; z → 1 ] ∧
        rotate(X, α, β, θ, $P_x$) ∧
        rotate(Y, α, β, θ, $P_y$) ∧
        rotate(Z, α, β, θ, $P_z$) ∧
        $V_x$ : vector [ a → $P_x$.x; b → $P_x$.y; c → $P_x$.z] ∧
        $V_y$ : vector [ a → $P_y$.x; b → $P_y$.y; c → $P_y$.z] ∧
        $V_z$ : vector [ a → $P_z$.x; b → $P_z$.y; c → $P_z$.z] ∧
```

Using the above predicate we can now define the predicate *newTP(U, S, T, P)* which is true when P is the result of translating a point U as described by the *csTranslationPoint* T for a connection surface S. This predicate is defined in definition 8.3.7.

Definition 8.3.7

```
newTP(U, S, T, P) ← U:point ∧ S : plane ∧
    T : csTranslationPoint ∧
    planeRF(S, X, Y, Z) ∧
    ( ( T[direction1 → ”left”] ∧
        $V_{left}$ : vector [ a → T.distance1 * Y.a;
                           b → T.distance1 * Y.b;
                           c → T.distance1 * Y.c ]) ∨
      ... ∨
      ( T[direction2 → ”right”] ∧
        $V_{left}$ : vector [ a → -T.distance2 * Y.a;
                           b → -T.distance2 * Y.b;
                           c → -T.distance2 * Y.c ])) ∧
    P : point [x → U.x + $V_{left}$.a + $V_{front}$.a + $V_{top}$.a;
               y → U.y + $V_{left}$.b + $V_{front}$.b + $V_{top}$.b;
               z → U.z + $V_{left}$.c + $V_{front}$.c + $V_{top}$.c ]
```

Finally we can define the predicate *connectionSurfacePos(A, C, P)* which is true when P is the position for a concept A connected over a connection surface relation to a concept C. This predicate is defined in definition 8.3.8.

Definition 8.3.8

```
connectionSurfacePos(A, C, P) ← A : concept ∧ C : concept ∧
        A [ connectedTo@C •→ R] ∧ R : connectionSurfaceRelation ∧
        R [sourceSurface → S; targetSurface → T ] ∧
        planeProjection($P_1$, S, A.position) ∧
        planeProjection($P_2$, T, C.position) ∧
        $V_1$ : vector [ a → $P_1$.x; b → $P_1$.y; c → $P_1$.z] ∧
        $V_2$ : vector [ a → $P_2$.x; b → $P_2$.y; c → $P_2$.z] ∧
        isDifference(V, $V_2$, $V_1$) ∧
        $P_3$ : point [ x → A.position.x + V.a;
                    y → A.position.y + V.b;
                    z → A.position.z + V.c ] ∧
        newTP($P_3$, T, R.sourceTP, P)
```

Note that the above predicate only checks the position for an object connected to some other object by means of a connection surface relation. On the other hand, we also need to verify the orientation of the connected objects. As we have seen in step 2, the orientation of the source object needs to be in such a way that its connection surface is parallel to the connection surface for the target object. Therefore we will define the predicate *connectionSurfaceOrient(A, C, E)* which is true when E is the external orientation for some concept A connected to a concept C by means of a connection surface relation. This predicate is defined in definition 8.3.9.

Definition 8.3.9

```
connectionSurfaceOrient(A, C, E) ← A : concept ∧ C : concept ∧ E : orientation ∧
        A [ connectedTo@C •→ R] ∧ R : connectionSurfaceRelation ∧
        R [sourceSurface → S; targetSurface → T ] ∧
        planeParameters(S, $N_s$, $P_s$) ∧
        planeParameters(T, $N_t$, $P_t$) ∧
        $A_s$ : line [$x_0$ → $P_s$.x; $y_0$ → $P_s$.y; $z_0$ → $P_s$.z;
                a → $N_s$.a; b → $N_s$.b; c → $N_s$.c ] ∧
        $A_t$ : line [$x_0$ → $P_t$.x; $y_0$ → $P_t$.y; $z_0$ → $P_t$.z;
                a → $N_t$.a; b → $N_t$.b; c → $N_t$.c ] ∧
        angleBetween($A_s$, $A_t$, $\gamma$) ∧
        $\alpha$ is 180 - $\gamma$ ∧
        isOrthogonal($A_s$, $A_t$, V) ∧
        eulerAngles($\beta$, $\sigma$, $\theta$, V, $\alpha$) ∧
        A[externalOrientation → $E_A$] ∧
        E : orientation[ frontAngle → $\beta$ + $E_A$.frontAngle;
```

leftAngle $\rightarrow \sigma$ + E_A*.leftAngle;*
topAngle $\rightarrow \theta$ + E_A*.topAngle]*

The only thing which lefts us to do is to extend the definitions of the *connectedPosition(A, C, P)* and the *connectedOrientation(A, C, E)* predicates. These predicates were already defined and extended to capture the initial semantics of the connection axis relation and connection point relation in respectively the *position* and *externalOrientation* properties for concepts. In definition 8.3.10 we give the extended definition for the *connectedPosition* predicate while in definition 8.3.11 we give the extended definition for the *connectedOrientation* predicate. Note that the *connectedPosition* predicate also takes a possible translation of the target object into account according to the target translation point specification.

Definition 8.3.10
connectedPosition(A, C, P) $\leftarrow$ *A : concept* $\wedge$ *C : concept* $\wedge$ *P : point* $\wedge$
    (A [connectedTo@C $\bullet\!\!\rightarrow$ *R]* $\wedge$ *R : connectionPointRelation* $\wedge$
      sourcePositionCP(A, C, R, P)) $\vee$
    (A [connectedTo@C $\bullet\!\!\rightarrow$ *R]* $\wedge$ *R : connectionAxisRelation* $\wedge$
      caPos(A, C, P)) $\vee$
    (C [connectedTo@A $\bullet\!\!\rightarrow$ *R]* $\wedge$ *R : connectionAxisRelation* $\wedge$
      TPPos(A, R, P))
    (A [connectedTo@C $\bullet\!\!\rightarrow$ *R]* $\wedge$ *R : connectionSurfaceRelation* $\wedge$
      connectedSurfacePos(A, C, P)) $\vee$
    (C [connectedTo@A $\bullet\!\!\rightarrow$ *R]* $\wedge$ *R: connectionSurfaceRelation* $\wedge$
      newTP(A.position, R.targetPlane, R.targetTP, P))

Definition 8.3.11
connectedOrientation(A, C, E) $\leftarrow$
    A : concept $\wedge$ *C : concept* $\wedge$ *E : orientation* $\wedge$
    A [connectedTo@C $\bullet\!\!\rightarrow$ *R]* $\wedge$
    ((R : connectionAxisRelation $\wedge$ *connectionAxisOrient(A, C, E))* $\vee$
      (R : connectionSurfaceRelation $\wedge$ *connectionSurfaceOrient(A, C, E)))*

Formalization of the simulation semantics

Also the simulation semantics for the connection surface relation are very similar to the simulation semantics for the connection axis relation. Where for the initial semantics, the source object needs to be positioned and oriented in such a way that the connection surface relation is respected, the simulation semantics needs to enforce that whenever one of the objects moves or changes its orientation, the other one may have to move or re-orient in order to respect the connection surface relation.

This will happen exactly in the same way as we have described in the first four steps. The fifth step is not needed since the translation points are only of importance to the initial position of the connected objects along the surface. However, in step 4 we translate the source so that its connection surface falls together with the connection surface of the target. This translation is equal to the vector from the source translation point to the target translation point. Since for the simulation semantics we don't have translation points, the translation of the source is equal to the distance between both connection surfaces. Therefore we will define the predicate *planePlaneDist(S, T, D)* which is true when D is the distance between two parallel planes S and T. This predicate is defined in definition 8.3.12.

Definition 8.3.12

planePlaneDist(S, T, D) $\leftarrow$
    S : plane $\wedge$ *T : plane* $\wedge$
    planeParameters(T, N, P) $\wedge$
    $D\ is\ \frac{|N.a*(S.x_0-T.x_0)+N.b*(S.y_0-T.y_0)+N.c*(S.z_0-T.z_0)|}{\sqrt{N.a^2+N.b^2+N.c^2}}$

Next we define the predicate *csPosConstraint(A, C, P)* which is true when P is the position of a concept A connected to a concept C by means of a connection surface relation. The difference between this predicate and the *connectionSurfacePos* predicate (see definition 8.3.8) is that concept A is translated over the distance between both connection surfaces instead of over the distance between the translation points. The *csPosConstraint* predicate is defined in definition 8.3.13.

Definition 8.3.13

csPosConstraint(A, C, P) $\leftarrow$ *A : concept* $\wedge$ *C : concept* $\wedge$
    ((*A [connectedTo@C* $\bullet\!\!\rightarrow$ *R]* $\wedge$
        R [sourceSurface $\rightarrow$ *S; targetSurface* $\rightarrow$ *T])* $\vee$
      (*C [connectedTo@A* $\bullet\!\!\rightarrow$ *R]* $\wedge$
        R [sourceSurface $\rightarrow$ *T; targetSurface* $\rightarrow$ *S])))* $\wedge$
    R : connectionSurfaceRelation $\wedge$
    planeParameters(T, N, P) $\wedge$
    $U\ is\ \sqrt{N.a^2 + N.b^2 + N.c^2}$ $\wedge$
    planePlaneDist(S, T, D) $\wedge$

$$
\begin{array}{ll}
V : \mathit{vector}\ [& a \rightarrow \frac{N.a*D}{U};\\
 & b \rightarrow \frac{N.b*D}{U};\\
 & c \rightarrow \frac{N.c*D}{U}\ ] \wedge\\
P : \mathit{point}\ [& x \rightarrow A.position.x + V.a;\\
 & y \rightarrow A.position.y + V.b;\\
 & z \rightarrow A.position.z + V.c\ ] \wedge
\end{array}
$$

Finally we can define the simulation semantics by using the above defined predicates. The simulation semantics for the connection surface relation is defined in definition 8.3.14.

Definition 8.3.14 *If two concepts a and b (thus a :: concept and b :: concept) are connected via a connection surface relation r (thus r : connectionSurfaceRelation), then for their simulation semantics the position and orientation of a and b must be as follows at all times:*

$csPosConstraint(a,\ b,\ P_a) \wedge a[position \rightarrow P_a]$
$csPosConstraint(b,\ a,\ P_b) \wedge b[position \rightarrow P_b]$

$connectionSurfaceOrient(a,\ b,\ E_a) \wedge a[externalOrientation \rightarrow E_a]$
$connectionSurfaceOrient(b,\ a,\ E_b) \wedge b[externalOrientation \rightarrow E_b]$

Chapter 9

Formalizing Constraints

9.1 Slider constraint

In this section we will formally define the slider constraint. As we have seen earlier, the slider constraint restricts two objects connected over a connection axis relation in such a way that they can only move over a certain distance along the axis.

First we will give the formal definition of the slider constraint. This is defined in definition 9.1.1. A *slider* has a property which keeps track of the connection axis relation the slider is defined on. It also has a property *onConcept* who's value is the concept on which the constraint is defined. Next, it has a *direction* property of which the value indicates one of the directions in which the object can move. Note that in the informal specification there are two directions (used for readability) but since the directions need to be opposite we don't need to keep them both inside the *slider* object. Finally a *slider* has the properties *distance*1 and *distance*2. The value of *distance*1 indicates the distance the object may move in the direction of property *direction*. *Distance*2 indicates how far the object may move in the other (opposite) direction. Note that *distance*1 and *distance*2 are also defined as inheritable properties with value ∞. This is because we informally defined that when the user does not specify limits for the slider constraint, then the constrained objects may move infinitely along the connection axis.

Definition 9.1.1
slider [CARelation $\Rightarrow$ *connectionAxisRelation;*
onConcept $\Rightarrow$ *concept;*
distance1 $\Rightarrow$ *float;*

$$\begin{aligned}
&\mathit{distance1} \bullet\!\!\rightarrow \infty;\\
&\mathit{distance2} \Rightarrow \mathit{float};\\
&\mathit{distance2} \bullet\!\!\rightarrow \infty;\\
&\mathit{direction} \Rightarrow \mathit{string}\ ]
\end{aligned}$$

Next we will define the semantics of the slider constraint. Note that constraints do not have an initial semantics; and therefore we only need to define the simulation semantics.

Formalization of the simulation semantics

First we will define a predicate *moveAxis(A, R, L)* which is true when L is a line parallel with the connection axis going through the position of a concept A, when A is involved a connection axis relation R either as source or as target object. This predicate is defined in definition 9.1.2.

Definition 9.1.2

$$\begin{aligned}
&\mathit{moveAxis(A,\ R,\ L)} \leftarrow \mathit{A : concept} \wedge\\
&\quad \mathit{R : connectionAxisRelation} \wedge \mathit{L : line} \wedge\\
&\quad \mathit{R\ [targetAxis} \rightarrow \mathit{M\ ]} \wedge\\
&\quad \mathit{L : line\ [}\ x_0 \rightarrow \mathit{A.position.x};\\
&\qquad\qquad y_0 \rightarrow \mathit{A.position.y};\\
&\qquad\qquad z_0 \rightarrow \mathit{A.position.z};\\
&\qquad\qquad a \rightarrow \mathit{M.a};\ b \rightarrow \mathit{M.b};\ c \rightarrow \mathit{M.c}\ ]
\end{aligned}$$

When an object involved in a connection axis relation is constrained via a slider constraint it is only allowed to move over its move axis (which is the axis through the position of the object parallel to the connection axis) over a certain distance. This way we can define the slider constraint using a predicate *sliderConstraint(A, S)* which needs to be true during the complete lifetime of the constrained object.

Definition 9.1.3

$$\begin{aligned}
&\mathit{sliderConstraint(A,\ S)} \leftarrow \mathit{A : concept} \wedge\\
&\quad (\mathit{A[connectedTo@B} \bullet\!\!\rightarrow \mathit{R]} \vee \mathit{B[connectedTo@A} \bullet\!\!\rightarrow \mathit{R]}) \wedge\\
&\quad \mathit{moveAxis(A,\ R,\ L)} \wedge\\
&\quad \mathit{projection(P,\ A.position,\ L)} \wedge\\
&\quad \mathit{parameterValue(L,\ P,\ T)} \wedge\\
&\quad \mathit{projection(Q,\ B.position,\ L)} \wedge\\
&\quad \mathit{parameterValue(L,\ Q,\ U)} \wedge
\end{aligned}$$

((S [direction → "left"] ∧ *(1)*
((L.b > 0 ∧ U-S.distance2 < T < U + S.distance1) ∨
(L.b < 0 ∧ U-S.distance1 < T < U+S.distance2))) ∨
(S [direction → "top"] ∧ *(2)*
((L.c > 0 ∧ U-S.distance2 < T < U + S.distance1) ∨
(L.c < 0 ∧ U-S.distance1 < T < U+S.distance2))) ∨
. . .)

Finally, definition 9.1.4 formally defines the simulation semantics for the slider constraint.

Definition 9.1.4 *Take a and b to be two concepts, thus a :: concept and b :: concept, connected via a connection axis relation r. Thus a[connectedTo@b •→ r]. Also take the following slider constraint defined on the connection axis relation r for the concept a (where "dir" is one of the known directions):*

s : slider [CARelation → r,
onConcept → a,
distance1 → D,
distance2 → E,
direction → "dir"]

Then the above slider constraint is respected when the following predicates are true during the complete lifetime of concept a:

sliderConstraint(a, S) sliderConstraint(b, S)

9.2 Hinge constraint

In this section we will formally define the hinge constraint. Remember that an object connected via a connection axis relation can be constrained by the hinge constraint in such a way that the object is only allowed to move a certain angle around the connection axis. First we will give the definition of the hinge constraint in definition 9.2.1.

Definition 9.2.1
hinge [CARelation ⇒ connectionAxisRelation;
onConcept ⇒ concept;
viewpoint ⇒ string;
cw ⇒ float;

cw $\bullet\!\!\rightarrow$ ∞*;*
ccw $\Rightarrow$ *float;*
ccw $\bullet\!\!\rightarrow$ ∞ *]*

So the hinge constraint has a property which contains the connection axis on which the hinge is defined and a property referring to the concept which needs to be constrained. It also has a viewpoint property indicating the viewpoint of the designer onto the connection axis. Next, the hinge constraint also has a *cw* and *ccw* property which contains the angle that the object may rotate around the connection axis in respectively the clockwise and counterclockwise directions. Note that *cw* and *ccw* are also inheritable with value ∞. Now we can define the simulation semantics for the slider constraint. Note that similar to the slider constraint we do not have initial semantics.

Formalization of the simulation semantics

Suppose that for a connection axis relation the source concept is constrained by means of a hinge constraint. Take L as the connection axis and p as the initial position of the source concept (see figure 9.1). The specified hinge constraint will only allow the source concept to be positioned on the circle shown in figure 9.1. Take c to be the center of this circle. Note that we can calculate c as the orthogonal projection of the position p of the source concept onto the connection axis L.

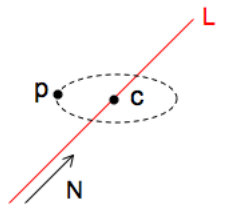


Figure 9.1: Semantics of the hinge constraint

So we have the following information:

- p is the position of the constrained object.
- L is the connection axis on which the hinge constraint is defined.

- C is the orthogonal projection of p on L. This means that c is the center of the circle on which the constrained object is allowed to move.
- N is a vector parallel to the connection axis L.
- U is a unit vector pointing from the center c towards the point p.
- R is the radius of the circle on which the constrained object is allowed to move. Thus R is the distance from c to p.
- V is the vector $N \times U$.

Using the above information we can now give the parametic equation of the circle on which the object is allowed to move. This parametric equation is as follows:

$$(x, y, z) = C + R\cos(t)U + R\sin(t)V$$

Now suppose that the constrained objects are allowed to rotate α degrees clockwise and β degrees counterclockwise around the connection axis. Now, depending on the viewpoint we can determine the possible values for the parameter t in the above equation. When the viewpoint (which is specified by the designer for the hinge constraint) is *left, top* or *front*, then $-\alpha \leq t \leq \beta$. When the viewpoint is *right, bottom* or *back*, then $-\beta \leq t \leq \alpha$.

Now we formalize the above theory by defining a predicate which is true when the position of the constrained object lies on the circle as described above. The simulation semantics is then defined by the fact that the defined predicate must be true during the complete lifetime of the constrained object.
We start with defining a predicate isElement(P, C, T) which is true when P is a point that lies on the circle C for a certain parameter value T in the equation of C. This predicate is defined in definition 9.2.2.

Definition 9.2.2
isElement(P, C, T) $\leftarrow$ *P : point* $\wedge$ *C : circle* $\wedge$
P [x $\rightarrow$ *C.c.x + r cos(T) C.u.a + r sin(T) C.v.a;*
y $\rightarrow$ *C.c.y + r cos(T) C.u.b + r sin(T) C.v.b;*
z $\rightarrow$ *C.c.z + r cos(T) C.u.c + r sin(T) C.v.c]*

Next we will define the predicate *moveCircle(H, C)* which is true when C is the cirlce on which an object constrained by the hinge constraint H is allowed to move. This predicate is defined in definition 9.2.3.

Definition 9.2.3

$moveCircle(H,\ C) \leftarrow H : hinge \wedge C : circle \wedge$
$\quad H\ [\ CARelation \rightarrow R\ ] \wedge$
$\quad A\ [\ connectedTo@B \bullet\!\!\rightarrow R\ ] \wedge$
$\quad R\ [\ targetAxis \rightarrow M\ ] \wedge$
$\quad projection(C_0,\ H.onConcept.position,\ M) \wedge$
$\quad N : vector\ [\ a \rightarrow M.a;\ b \rightarrow M.b;\ c \rightarrow M.c\ ]$
$\quad isDifference(U,\ H.onConcept.position,\ C_0) \wedge$
$\quad R\ is\ \sqrt{U.a_2 + U.b_2 + U.c_2} \wedge$
$\quad U_1 : vector\ [\ a\rightarrow \frac{U.a}{R};\ b\rightarrow \frac{U.b}{R};\ c\rightarrow \frac{U.c}{R}\ ] \wedge$
$\quad V : vector\ [\ a \rightarrow N.b\ U.c\ \text{-}\ N.c\ U.b;$
$\qquad\qquad b \rightarrow N.c\ U.a\ \text{-}\ N.a\ U.c;$
$\qquad\qquad c \rightarrow N.a\ U.b\ \text{-}\ N.b\ U.a\ ] \wedge$
$\quad C : circle\ [c \rightarrow C_0;\ r \rightarrow R;\ u \rightarrow U_1;\ v \rightarrow V]$

The last predicate we will define is the predicate *hingeConstraint(H)* which is true when all conditions for a hinge constraint H are satisfied. This predicate is defined in definition 9.2.4.

Definition 9.2.4

$hingeConstraint(H) \leftarrow H : hinge \wedge$
$\quad H\ [\ onConcept \rightarrow A\ ;\ cw \rightarrow \alpha;\ ccw \rightarrow \beta] \wedge$
$\quad moveCircle(H,\ C) \wedge$
$\quad isElement(A.position,\ C,\ T) \wedge$
$\quad (\ (\ (\ H[viewpoint \rightarrow \text{"left"}] \vee$
$\qquad\quad H[viewpoint \rightarrow \text{"top"}] \vee$
$\qquad\quad H[viewpoint \rightarrow \text{"front"}]\ ) \wedge$
$\qquad (\ \text{-}\alpha \leq\ T \leq \beta)) \vee$
$\qquad (\ (\ H[viewpoint \rightarrow \text{"right"}] \vee$
$\qquad\quad H[viewpoint \rightarrow \text{"bottom"}] \vee$
$\qquad\quad H[viewpoint \rightarrow \text{"back"}]\ ) \wedge$
$\qquad (\ \text{-}\beta \leq\ T \leq \alpha)))$

Finally we can define the simulation semantics for the hinge constraint. This is done in definition 9.2.5.

Definition 9.2.5 *Take a and b to be two concepts, thus a :: concept and b :: concept, connected by means of a connection axis relation r. Also take a to be constrained by means of the hinge constraint h defined as follows:*

h [CARelation → r;
onConcept → a]

Then the simulation semantics for the hinge constraint means that the following predicate must always be true during the complete lifetime of object a:

hingeConstraint(h)

The above definition completes the formal definition for the simulation semantics of the hinge constraint.

9.3 Joystick constraint

In this section we will formally define the joystick constraint. Remember that the joystick constraint allows two objects connected by means of a connection point relation to rotate over a certain angle around two perpendicular axes through the connection point. First we will give the definition of the joystick constraint. The joystick constraint is formally defined in definition 9.3.1.

Definition 9.3.1
joystick [CPRelation ⇒ connectionPointRelation;
onConcept ⇒ concept;
axis1 ⇒ axis;
axis2 ⇒ axis;
angle1 ⇒ float;
angle2 ⇒ float;
rotationAxis ⇒ string;
rotationAngle ⇒ float]

So a joystick constraint has the following properties:

- **CPRelation**: contains the connection point relation on which the joystick constraint is defined.
- **onConcept**: contains the concept (source or target of the connection point relation) on which the joystick constraint is defined.
- **axis1 and angle1**: describes one of both perpendicular axes around which the object is allowed to rotate angle1 degrees in the clockwise as well as the counterclockwise directions.

- **axis2, angle2**: describes the other of both perpendicular axes around which the object is allowed to rotate angle2 degrees in the clockwise as well as the counterclockwise directions.

- **rotationAxis, rotationAngle**: specifies a possible rotation for the perpendicular axes $axis1$ and $axis2$. As the designer can use three default axes (left-to-right, top-to-bottom and front-to-back), a rotation of these default axes offers more possibilities, as we have discussed in chapter 4.

So an object constrained by means of a joystick constraint (see figure 9.2(a)) is actually allowed to move as if it was inside a cone (illustrated in figure 9.2(b)). So we will need to calculate the equation of the elliptical cone for a specific joystick constraint. Next, we need to define a predicate that is true when a point P falls inside an elliptical cone C. Then the simulation semantics for the joystick constraint will be defined by requiring that the position of the constrained object falls inside the cone corresponding to the joystick constraint.

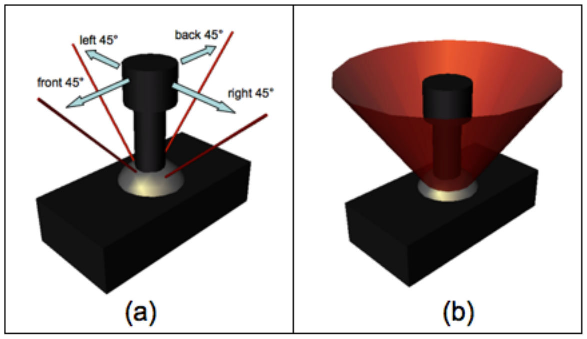


Figure 9.2: Example of a joystick constraint

Take the cone as shown in figure 9.3. This cone opens towards the z-direction and has its vertex at (x_0, y_0, z_0). All points which fall inside or on the cone are described by the following parametric equations (for $\theta \in [0, 2\pi[$ and u $\in$ [0, h]):

$$\begin{cases} x = x_0 + (a-s)\frac{h-u}{h}\cos\theta & 0 \le s \le a \\ y = y_0 + (b-t)\frac{h-u}{h}\sin\theta & 0 \le t \le b \\ z = z_0 + h - u \end{cases}$$

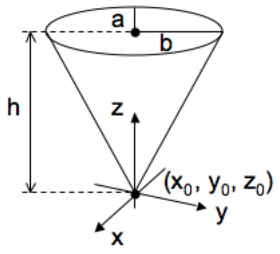


Figure 9.3: Example of a cone opening towards the z-direction

So using the above equation we can check whether a point falls inside or on a given elliptical cone. Next we need to find a way to calculate the cone corresponding to a joystick constraint. By the specification of the joystick constraint we have the following information: the connection point $p_0 = (x_0, y_0, z_0)$ and the position $p_1 = (x_1, y_1, z_1)$of the constrained object. Thus in order to have the equation of the corresponding cone we need to calculate the height h of the cone, the semiminor axis a and the semimajor axis b.

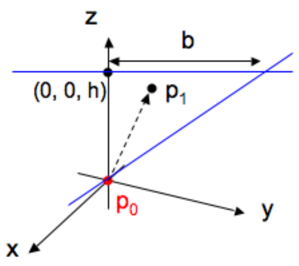


Figure 9.4: Calculation of the cone

We can calculate the height of the cone by calculating the distance from the point p_0 to p_1. This will be the maximum height a point inside the cone will be able to take. Next we need to calculate the semimajor axis b. Take L to be the line through the point (0, 0, h) parallel to the y-axis. Next calculate the vector v as being the vector (0, 0, 1) rotated around axis1 of the hinge constraint (in fig. 9.4 this is the x-axis) with angle1 degrees (which is α in our example). Then take M to be the line throught p_0 parallel to the vector v. If we now take the intersection of L and M and substract the point (0, 0,

h) from it we get the semimajor axis b. It is clear thet the semiminor axis a can be calculated similarly.

Now we will formally define this in F-logic. Inside the formal definition we will also take into account a possible rotation of the perpendicular axes of the hinge constraint (which is a simple orientation) and we will also take into account cones which open towards the x-direction or the y-direction.

Formalization of the simulation semantics

First we will define three subclasses of the class cone (see chapter 6). We will first start with the class *zCone* which represents a cone that opens towards the Z-direction. This class is defined in definition 9.3.2.

Definition 9.3.2
zCone [xyz@s, t, u, θ ⇒ point]

zCone :: cone

where xyz is a method defined as follows:

$$
\begin{aligned}
&C[xyz@s,\ t,\ u,\ \theta \rightarrow P] \leftarrow C : cone \wedge \\
&\qquad C\ [\ vertex \rightarrow Q;\ height \rightarrow h; \\
&\qquad\quad semiminor \rightarrow a;\ semimajor \rightarrow b; \\
&\qquad\quad rotationAxis \rightarrow V; \\
&\qquad\quad rotationAngle \rightarrow \alpha] \wedge \\
&\qquad P_1 : point\ [\ x \rightarrow Q.x + (a\text{-}s)\tfrac{h-u}{h}\ \cos\theta; && (1) \\
&\qquad\qquad\quad y \rightarrow Q.y + (a\text{-}s)\tfrac{h-u}{h}\ \sin\theta; && (2) \\
&\qquad\qquad\quad z \rightarrow Q.z + h - u\ ] \wedge && (3) \\
&\qquad eulerAngles(\beta,\ \sigma,\ \gamma,\ V,\ \alpha) \wedge \\
&\qquad rotate(P_1,\ \beta,\ \sigma,\ \gamma,\ P)
\end{aligned}
$$

So the method *xyz* returns a point inside or on the cone for the given cone parameters s, t, u and θ.

Similar we can define the classes *xCone* and *yCone* as subclasses of the class *cone*. These classes will represent respectively cones which open towards the X-direction or towards the Y-direction. The definition of the *xyz* method will be similar to the one we have given in definition 9.3.2. The only difference will be in the lines (1)-(3). For example, for the xCone, the x property of p_1 will be equal to $Q.x + h - u$ while the z property will be equal to $Q.z + (a - s)\frac{h-u}{h}cos\theta$.

Next we will introduce the predicate *constraintZCone(J, H, P, C)*. This predicate is true when C is a zCone with height H and vertex P corresponding with the joystick constraint J. This predicate is defined in definition 9.3.3.

Definition 9.3.3

```
constraintZCone(J, H, P, C) ← J : joystick ∧ P : point ∧
        Q : point [ x → 0; y → 0; z → 1 ] ∧
        ( ( J [ axis1 → "frontToBack" ] ∧
            rotate(Q, $\frac{J.angle1}{2}$, 0, 0, $V_1$) ∧
            rotate(Q, 0, $\frac{J.angle2}{2}$, 0, $V_2$)) ∨
          ( J[ axis1 → "leftToRight"] ∧
            rotate(Q, $\frac{J.angle2}{2}$, 0, 0, $V_1$) ∧
            rotate(Q, 0, $\frac{J.angle1}{2}$, 0, $V_2$))) ∧
        $T_1$ is $\frac{H}{V_1.c}$ ∧
        $X_1$ is $T_1$ * $V_1.a$ ∧ $Y_1$ is $T_1$ * $V_1.b$ ∧
        $T_2$ is $\frac{H}{V_2.c}$ ∧
        $X_2$ is $T_2$ * $V_2.a$ ∧ $Y_2$ is $T_2$ * $V_2.b$ ∧
        B is $\sqrt{X_1^2 + Y_1^2}$ ∧
        A is $\sqrt{X_2^2 + Y_2^2}$ ∧
        C : zCone [ vertex → P;
                    height → H;
                    semiminor → A;
                    semimajor → B;
                    rotationAxis → J.rotationAxis;
                    rotationAngle → J.rotationAngle]
```

Similar, we can define predicates which are true when C is an xCone or a yCone with height H and vertex P corresponding with the joystick constraint J. These predicates are named *constraintXCone* and *constraintZCone*. Now we can define the predicate *constraintCone(J, C)* which is true when C is the cone corresponding to a joystick constraint J. This predicate is defined in definition 9.3.4.

Definition 9.3.4

```
constraintCone(J, C) ← J : joystick ∧
        A[ connectedTo@B → J.CPRelation] ∧
```

targetCPPosition(B, J.CPRelation, P_0*)* $\wedge$
J [onConcept $\rightarrow$ *C[position* $\rightarrow$ P_1*]]* $\wedge$
H is $\sqrt{(P_1.x - P_0.x)^2 + (P_1.y - P_0.y)^2 + (P_1.z - P_0.z)^2}$
(((J [axis1 $\rightarrow$ *"frontToBack"]* $\wedge$ *J [axis2* $\rightarrow$ *"leftToRight"])* $\vee$
(J [axis1 $\rightarrow$ *"leftToRight"]* $\wedge$ *J[axis2* $\rightarrow$ *"frontToBack])* $\wedge$
constraintZCone(J, H, P_0*, C))* $\vee$
((J [axis1 $\rightarrow$ *"frontToBack"]* $\wedge$ *J [axis2* $\rightarrow$ *"topToBottom"])* $\vee$
(J [axis1 $\rightarrow$ *"topToBottom"]* $\wedge$ *J[axis2* $\rightarrow$ *"frontToBack])* $\wedge$
constraintYCone(J, H, P_0*, C))* $\vee$
((J [axis1 $\rightarrow$ *"leftToRight"]* $\wedge$ *J [axis2* $\rightarrow$ *"topToBottom"])* $\vee$
(J [axis1 $\rightarrow$ *"topToBottom"]* $\wedge$ *J[axis2* $\rightarrow$ *"leftToRight])* $\wedge$
constraintZCone(J, H, P_0*, C)))*

Finally using the above predicates we can now define a predicate *joystick-Constraint(J)* which is true when there exist values for the parameters s, t, u and σ in the equation of the cone corresponding to the joystick constraint such that the position of the onConcept property is the value of the xyz property for the cone. With other words, this predicate is true when the position of the onConcept property value falls inside or on the cone corresponding to the joystick constraint. This predicate is defined in definition 9.3.5.

Definition 9.3.5
joystickconstraint(J) $\leftarrow$ *J : joystick* $\wedge$
J [onConcept $\rightarrow$ *A]* $\wedge$
constraintCone(J, C) $\wedge$
$\sigma \geq 0 \wedge \sigma \leq 2\Pi \wedge$
$u \geq 0 \wedge u \leq$ *C.height* $\wedge$
$s \geq 0 \wedge s \leq$ *C.semiminor* $\wedge$
$t \geq 0 \wedge t \leq$ *C.semimajor* $\wedge$
C[xyz@s, t, u, $\sigma \rightarrow$ *A.position]*

Using the above predicate we will now define the simulation semantics for the joystick constraint. This is defined in definition 9.3.6.

Definition 9.3.6 *Take a and b to be two concepts, thus a :: concept and b :: concept, connected by means of a connection point relation r. Also take a to be constrained by means of the joystick constraint j defined as follows:*

j [CPRelation $\bullet\!\!\rightarrow$ *r;*
onConcept $\rightarrow$ *a]*

Then the simulation semantics for the joystick constraint means that the following predicate must always be true during the complete lifetime of object a:

joystickConstraint(j)

9.4 Fixed Relative Position Constraint

Remember that the fixed relative position constraint enforces two objects to keep the same position relative to each other during their complete lifetime. This constraint can be specified on top of a spatial relation or it can hold between two objects for which no spatial relation has been specified. First we will formalize the fixed relative position constraint when specified on top of an explicit spatial relation between two objects. Next we will give the formalization for a fixed relative position constraint between two objects that have no explicit spatial relation in common.

In case of an explicit spatial relation

The fixed relative position constraint is defined in definition 9.4.1. It has one attribute, namely *onSpatialRel* representing the spatial relation on which the fixed relative position constraint is defined.

Definition 9.4.1
fixedRelativePositionSR [onSpatialRel ⇒ spatialRelation]

Suppose that between the two objects a and b a spatial relation has been defined. Let's say a is positioned left of b with a distance d. As we have seen before, this spatial relation is formalized on the initial semantics level (see chapter 7). If the designer has specified a fixed relative position constraint on top of the spatial relation between a and b then this means that this spatial relation not only holds on the initial semantics level but also on the simulation semantics level. In order to respect the fixed relative position constraint we need to add the following fact in our simulation semantics level:

$a[position \rightarrow P] \leftarrow leftOfPos(b, d, P)$

Actually, the above fact overwrites the position method for instance a. It means that the position of a must be left of b with distance d. It is easy to see that the formalization of the fixed relative position constraint on top of all other spatial relations is similar.

In case there is no spatial relation involved

When there is no spatial relation involved, the fixed relative position constraint is defined as in definition 9.4.2. It has two attributes, namely *concept1* and *concept2* representing the concepts between which the fixed relative position constraint is defined.

Definition 9.4.2
fixedRelativePosition [concept1 ⇒ concept;
concept2 ⇒ concept]

Suppose object a is defined to have a fixed relative position according to an object b. This means that a needs to keep the same relative position to b according to the initial situation. Therefore from the initial semantics level we need to calculate the distance from a to b in all possible directions (left-right, top-bottom and front-right). These distances need to be maintained during the rest of the life cycle of a and b. Suppose we define the distances in all directions between a and b from the initial semantics level as follows:

$xDiff$ is $a.position.x - b.position.x$
$yDiff$ is $a.position.y - b.position.y$
$zDiff$ is $a.position.z - b.position.z$

Then the fixed relative position constraint between a and b can be formalized in the simulation semantics level by adding the following deductive rule:

$a[position \rightarrow P] \leftarrow RelativePosition(b, xDiff, yDiff, zDiff, P)$

Remember from chapter 7 that the predicate RelativePosition (B, X, Y, Z, P) is true when P is the position that lies X units in front of, Y units left of and Z units on top of some object B.

9.5 Fixed Relative Orientation Constraint

The fixed relative orientation constraint freezes one object's orientation relative to another object's orientation. This constraint can be specified on top of an orientation relation or it can hold between two objects for which no orientation relation has been specified.

In case of an explicit orientation relation

The fixed relative orientation constraint is defined in definition 9.5.1. It has one attribute, namely *onOrientationRel* representing the relative orientation relation on which the fixed relative position orientation is defined.

Definition 9.5.1
fixedRelativeOrientationOR [onOrientationRel $\Rightarrow$ *relativeOrientationRelation]*

Suppose that between the two objects a and b an orientation relation has been defined. Let's say a is oriented with its left side towards the left side of b. This orientation relation is formalized on the initial semantics level (see chapter 7). If the designer specifies a fixed relative orientation constraint on top of the orientation relation between a and b then this means that this orientation relation not only holds on the initial semantics level but also on the simulation semantics level. In order to respect the fixed relative orientation constraint on top of the specified orientation relation we need to add the orientation relation to our simulation semantics level as follows:

$$\begin{aligned}&a[oriented@b{\bullet\!\!\rightarrow}\ r]\wedge\\ &r : relativeOrientationRelation[sourceFace \rightarrow \text{"}front\text{"};\\ &\qquad\qquad\qquad\qquad\qquad\qquad targetFace \rightarrow \text{"}front\text{"}]\end{aligned}$$

In case there is no orientation relation involved

When there is no orientation relation involved, the fixed relative orientation constraint is defined as in definition 9.5.2. It has two attributes, namely *concept1* and *concept2* representing the concepts between which the fixed relative orientation constraint is defined.

Definition 9.5.2
fixedRelativeOrientation [concept1 $\Rightarrow$ *concept;*
concept2 $\Rightarrow$ *concept]*

Suppose object a is defined to have a fixed relative orientation according to an object b. This means that when b changes it's orientation, a will undergo the same change in it's orientation. Therefore from the initial semantics level we need to calculate the difference in orientation between a and b for the leftAngle, topAngle and frontAngle property of the orientations.

These differences in the external orientation of a and b need to be maintained during the rest of a and b their lifecycle. Suppose we define the differences in all orientations between a and b from the initial semantics level as follows:

$frontAngleDiff$ is
  $a.externalOrientation.frontAngle - b.externalOrientation.frontAngle$
$leftAngleDiff$ is
  $a.externalOrientation.leftAngle - b.externalOrientation.leftAngle$
$topAngleDiff$ is
  $a.externalOrientation.topAngle - b.externalOrientation.topAngle$

Then the fixed relative orientation constraint between a and b can be formalized in the simulation semantics level by adding the following deductive rule:

$a[externalOrientation \rightarrow E] \leftarrow$
  $bFrontAngle$ is $b.externalOrientation.frontAngle \wedge$
  $bLeftAngle$ is $b.externalOrientation.leftAngle \wedge$
  $bTopAngle$ is $b.externalOrientation.topAngle \wedge$
  $E : orientation[frontAngle \rightarrow bFrontAngle - frontAngleDiff;$
    $leftAngle \rightarrow bLeftAngle - leftAngleDiff;$
    $topAngle \rightarrow bTopAngle - topAngleDiff]$

The above deductive rule actually states that the external orientation of a must stay constant with respect to the external orientation of b according to the initial situation.

9.6 Positioning Constraint

The positioning constraint constrains the possibilities to place certain objects in the virtual environment. In contrast to all other connections and constraints defined so far, the positioning constraint only acts on the simulation semantics level. As we have seen in chapter 4, a positioning constraint is defined by means of an anchor area and a binding area. An object with an anchor area with a certain label can only be positioned in such a way that its anchor area falls inside the binding area (with the same label) of another object. However, these anchor and binding area's depend on the way an object is represented in the virtual environment. Therefore we first need to define what we mean by shape. This is done in definition 9.6.1. A *shape* is represented by means of an F-logic class having a *points* property containing all points of which the shape consists. The class *point* has been defined in definition 6.1.1 (see chapter 6). Furthermore, a shape also has a height, depth and width property defined as a method. For example, the height is the result of the z-value of the highest point of the shape minus the z-value of the lowest point of the shape. This follows from our definition for the default orientation of an object (see chapter 6). Next, a shape has a *top*, *bottom*, *left*, *right*, *front* and *back* property who's values are a set of points respectively representing the shape's top, bottom, left, right, front and back side. These properties are also defined by methods. Note that all of these methods are defined similarly. Therefore, we will only illustrate their definitions by defining the top and right property.

Definition 9.6.1 *A shape is defined as follows:*

$$
\begin{array}{l}
\mathit{shape[points} \Rightarrow\!\!\!\!> \mathit{point;}\\
\quad \mathit{height} \Rightarrow \mathit{float;}\\
\quad \mathit{depth} \Rightarrow \mathit{float;}\\
\quad \mathit{width} \Rightarrow \mathit{float;}\\
\quad \mathit{top} \Rightarrow\!\!\!\!> \mathit{point;}\\
\quad \mathit{bottom} \Rightarrow\!\!\!\!> \mathit{point;}\\
\quad \mathit{front} \Rightarrow\!\!\!\!> \mathit{point;}\\
\quad \mathit{back} \Rightarrow\!\!\!\!> \mathit{point;}\\
\quad \mathit{left} \Rightarrow\!\!\!\!> \mathit{point;}\\
\quad \mathit{right} \Rightarrow\!\!\!\!> \mathit{point]}
\end{array}
$$

$$
\begin{array}{l}
S[height \rightarrow H] \leftarrow S : shape \wedge\\
\quad (S[points \twoheadrightarrow P_1] \wedge \neg(S[points \twoheadrightarrow Q_1] \wedge Q_1.z > P_1.z))\\
\quad (S[points \twoheadrightarrow P_2] \wedge \neg(S[points \twoheadrightarrow Q_2] \wedge Q_2.z < P_2.z))\\
\quad H \ is \ |P_1.z \text{ - } P_2.z|
\end{array}
$$

$$
\begin{array}{l}
S[width \rightarrow W] \leftarrow S : shape \wedge\\
\quad (S[points \twoheadrightarrow P_1] \wedge \neg(S[points \twoheadrightarrow Q_1] \wedge Q_1.y > P_1.y))\\
\quad (S[points \twoheadrightarrow P_2] \wedge \neg(S[points \twoheadrightarrow Q_2] \wedge Q_2.y < P_2.y))\\
\quad W \ is \ |P_1.y \text{ - } P_2.y|
\end{array}
$$

$$
\begin{array}{l}
S[depth \rightarrow D] \leftarrow S : shape \wedge\\
\quad (S[points \twoheadrightarrow P_1] \wedge \neg(S[points \twoheadrightarrow Q_1] \wedge Q_1.x > P_1.x))\\
\quad (S[points \twoheadrightarrow P_2] \wedge \neg(S[points \twoheadrightarrow Q_2] \wedge Q_2.x < P_2.x))\\
\quad D \ is \ |P_1.x \text{ - } P_2.x|
\end{array}
$$

$$S[top \twoheadrightarrow P] \leftarrow S : shape \wedge$$
$$S[points \twoheadrightarrow P] \wedge P[z \rightarrow \frac{S.height}{2}]$$

$$S[right \twoheadrightarrow P] \leftarrow S : shape \wedge$$
$$S[points \twoheadrightarrow P] \wedge P[y \rightarrow \frac{-S.width}{2}]$$

Note that in order to calculate the points which belong to one of the surfaces, we assume that the shape is positioned at (0,0,0). The real position will be determined by the object mapped onto the shape. We will take this into account when defining the simulation semantics of the positioning constraint.
Now we will also formalize the mapping. We see a *mapping* as an F-logic class with two properties, a $fromConcept$ property containing the concept that will be mapped and a *toShape* property containing the shape to which the concept is mapped. This definition is given in definition 9.6.2.

Definition 9.6.2

$$mapping[fromConcept \Rightarrow concept,$$
$$toShape \Rightarrow shape]$$

Now we will define the anchor and binding constraints. The anchor constraint is defined in definition 9.6.3. While the *surface* property's value represents the name of the surface used to specify the anchor area, the *pointSurface* contains all points which belong to the anchor area. These points are calculated from the shape on which the object is mapped and takes the objects position and orientation into account.

Definition 9.6.3

$$anchor[\ onConcept \Rightarrow concept;$$
$$surface \Rightarrow string;$$
$$pointSurface \Rrightarrow point;$$
$$label \Rightarrow string]$$

where pointSurface is defined as follows:

$A[pointSurface \twoheadrightarrow P] \leftarrow A : anchor \wedge$
$\quad A[onConcept \rightarrow C] \wedge$
$\quad M : mapping \wedge M[fromConcept \rightarrow C;\ toShape \rightarrow S] \wedge$
$\quad C[position \rightarrow P_c] \wedge$
$\quad eFrontAngle(A, \alpha) \wedge eLeftAngle(A, \beta) \wedge eTopAngle(A, \gamma)$
$\quad (\ (A[surface \rightarrow 'top'] \wedge S[top \twoheadrightarrow Q]) \vee$
$\quad\ \ (A[surface \rightarrow 'bottom'] \wedge S[bottom \twoheadrightarrow Q]) \vee$
$\quad\ \ (A[surface \rightarrow 'front'] \wedge S[front \twoheadrightarrow Q]) \vee$
$\quad\ \ (A[surface \rightarrow 'back'] \wedge S[back \twoheadrightarrow Q]) \vee$
$\quad\ \ (A[surface \rightarrow 'left'] \wedge S[left \twoheadrightarrow Q]) \vee$
$\quad\ \ (A[surface \rightarrow 'right'] \wedge S[right \twoheadrightarrow Q])) \wedge$
$\quad R : point[x \rightarrow Q.x + P_c.x;$
$\qquad\qquad y \rightarrow Q.y + P_c.y;$
$\qquad\qquad z \rightarrow Q.z + P_c.z] \wedge$
$\quad rotate(R, \alpha, \beta, \gamma, P)$

Next, definition 9.6.4 defines the concept *binding*. Note that the *pointSurface* property is almost identically defined as in definition 9.6.3. Therefore we will not repeat this definition.

Definition 9.6.4

$binding[\ onConcept \Rightarrow concept;$
$\qquad surface \Rightarrow string;$
$\qquad pointSurface \Rrightarrow point;$
$\qquad label \Rightarrow string]$

Now that we have defined the anchor and binding area concepts we can define their semantics. A positioning constraint is satisfied when for an object A having an anchor area, there is some object B having a binding area with the same label as A's anchor area, so that at least one point of A's anchor area falls inside B's binding area. This is defined in definition 9.6.5. When the predicate defined in definition 9.6.5 is true for an object A having an anchor area, then the positioning constraint is satisfied.

Definition 9.6.5

positioningConstraint(A) $\leftarrow$ *A : concept* $\wedge$
N : anchor $\wedge$ *N[onConcept* $\rightarrow$ *A; label* $\rightarrow$ *L]* $\wedge$
B : concept $\wedge$
M : binding $\wedge$ *M[onConcept* $\rightarrow$ *B; label* $\rightarrow$ *L]* $\wedge$
P : point $\wedge$
N[pointSurface $\twoheadrightarrow$ *P]* $\wedge$
M[pointSurface $\twoheadrightarrow$ *P]*

Chapter 10

Formalizing CSG Relations

In the previous chapter, we informally introduced the modeling of complex shapes by means of CSG relations. Therefore we introduced three relations, namely the union relation, the intersection relation and the difference relation. In this section we will formalize these relations.

In contrast to the relations we formalized so far, the CSG relations create new shapes on which the related concepts or instances are mapped in the virtual environment rather than creating new concepts and instances from existing ones. In the following section the CSG relations are formalized.

10.1 Union Relation

A union relation between two objects a and b creates a new shape which exists of all points of the shape on which object a is mapped and of all points of the shape on which object b is mapped. Since a union relation actually defines a new shape we will formalize it as a subclass of *shape*. The *union* class will also have two additional properties, namely *concept*1 and *concept*2 of type *concept*. These properties represent the concepts between which a union relation is defined. Next, we also overwrite the *points* property. This property is now a method returning all points on which *concept*1 has been mapped and all points on which *concept*2 has been mapped. The union relation is formally defined by definition 10.1.1.

Definition 10.1.1

union :: shape
union[concept1 $\Rightarrow$ *concept,*

$$\textit{concept2} \Rightarrow \textit{concept}]$$

where the points property is overwritten with the following method signature:

$$\begin{aligned} A[\textit{points} \twoheadrightarrow P] \leftarrow\ & A : \textit{union} \wedge \\ & A[\textit{concept1} \rightarrow C_1, \\ & \quad \textit{concept2} \rightarrow C_2] \wedge \\ & M_1 : \textit{mapping}\ [\textit{fromConcept} \rightarrow C_1, \\ & \qquad\qquad \textit{toShape} \rightarrow S_1] \wedge \\ & M_2 : \textit{mapping}\ [\textit{fromConcept} \rightarrow C_2, \\ & \qquad\qquad \textit{toShape} \rightarrow S_2] \wedge \\ & (S_1[\textit{points} \twoheadrightarrow P] \vee S_2[\textit{points} \twoheadrightarrow P]) \end{aligned}$$

10.2 Intersection relation

An intersection relation between two objects a and b creates a new shape which exists of all points that belong to the shape on which object a has been mapped and also belong to the shape on which object b has been mapped. Again we will formally define an intersection relation as a subclass of *shape*. The definition for the intersection relation is very similar to the one we have given for the union relation. The only difference is in the *points* method which now returns all points that belong to the shape on which *concept*1 is mapped and also belong to the shape on which *concept*2 has been mapped. Definition 10.2.1 formally defines the intersection relation.

Definition 10.2.1

$$\begin{aligned} & \textit{intersection} :: \textit{shape} \\ & \textit{intersection}[\textit{concept1} \Rightarrow \textit{concept}, \\ & \qquad\qquad \textit{concept2} \Rightarrow \textit{concept}] \end{aligned}$$

where the points property is overwritten with the following method signature:

$$\begin{aligned} A[\textit{points} \twoheadrightarrow P] \leftarrow\ & A : \textit{intersection} \wedge \\ & A[\textit{concept1} \rightarrow C_1, \\ & \quad \textit{concept2} \rightarrow C_2] \wedge \\ & M_1 : \textit{mapping}\ [\textit{fromConcept} \rightarrow C_1, \\ & \qquad\qquad \textit{toShape} \rightarrow S_1] \wedge \\ & M_2 : \textit{mapping}\ [\textit{fromConcept} \rightarrow C_2, \\ & \qquad\qquad \textit{toShape} \rightarrow S_2] \wedge \end{aligned}$$

$(S_1[points \twoheadrightarrow P] \wedge S_2[points \twoheadrightarrow P])$

10.3 Difference relation

A difference relation between two objects a and b creates a new shape that exists of all points which belong to the shape on which object a has been mapped and do not belong to the shape on which object b has been mapped. The difference relation is formally defined in a similar way as the union and intersection relation. However, here there is one difference. While the union and intersection are symmetric, the difference relation is not. The difference relation is given in definition 10.3.1.

Definition 10.3.1

difference :: shape
difference[concept1 ⇒ concept,
concept2 ⇒ concept]

where the points property is overwritten with the following method signature:

$$\begin{aligned} A[points \twoheadrightarrow P] \leftarrow\ & A : difference \wedge \\ & A[concept1 \rightarrow C_1, \\ & \quad concept2 \rightarrow C_2] \wedge \\ & M_1 : mapping\ [fromConcept \rightarrow C_1, \\ & \qquad toShape \rightarrow S_1] \wedge \\ & M_2 : mapping\ [fromConcept \rightarrow C_2, \\ & \qquad toShape \rightarrow S_2] \wedge \\ & (S_1[points \twoheadrightarrow P] \wedge \neg(S_2[points \twoheadrightarrow P])) \end{aligned}$$

Chapter 11

Applications of the formalization

In the previous chapters we have given a complete formal specification of a set of high-level modeling concepts that can be used for specifying simple as well as complex VR objects on a higher level of abstraction. These modeling concepts are specified in an unambiguous way by means of the above formalization. In this chapter we will illustrate some of the possible applications of the formalization. In section 11.1 we will illustrate the use of F-logic to reason over the formal conceptual specifications. In section 11.2 we will illustrate how consistency checking can be performed on formal conceptual specifications.

11.1 Reasoning

In the beginning of this chapter we also stated that this formal specification allows us to reason about conceptual specification. In this section we will illustrate this by means of some examples.

Suppose we have an F-Logic knowledge base containing the conceptual specification for a certain virtual environment. Using a tool like Flora2 or OntoBroker we can perform some reasoning about this conceptual specification by feeding some queries to the reasoner. Let's look to a first example. Suppose we would like to know which instances (of either which concept) are defined to be left of an instance *myCar* of the concept *car*. As we have seen in definition 7.5.2 an object a is defined as being left of an object b as a[positioned@b → r] where r : spatialRelation[direction → "left"; distance

$\rightarrow$ d]. Take the following query:

$$\begin{aligned}&?{-}X : concept \wedge X[positioned@myCar \rightarrow R]\wedge\\&\quad r[direction \rightarrow "left"; distance \rightarrow D]\end{aligned}$$

A reasoning tool will return all objects which are explicitly specified as being left of the instance myCar as the answer to the above query. The answer will also contain the distance D for each object X. However, the above query only returns the objects that are explicitly modeled as being left of myCar. It is also possible that there are objects which are left of myCar but which aren't explicitly modeled as such. To retrieve these objects we can query the conceptual knowledge base using the following query:

$$\begin{aligned}&?{-}X : concept \wedge X[position \rightarrow P]\wedge\\&\quad relativePosition(myCar, 0, D, 0, P) \wedge D > 0\end{aligned}$$

The above query will return all objects which are positioned on a positive distance D towards the left of myCar. Now suppose we would like to know if there is an object which lies exactly 2 units left of myCar, then we could use the following query:

$$\begin{aligned}&?{-}X : concept \wedge X[position \rightarrow P]\wedge\\&\quad relativePosition(myCar, 0, 2, 0, P)\end{aligned}$$

The above queries show how the formal specification of a conceptual model can be used to reason about that conceptual model. It is also possible to provide the user with a number of predicates that can be used to query the system. We could for example define the following predicate:

$$\begin{aligned}objectsLeftOf(A, X) \leftarrow &X : concept \wedge A : concept\wedge\\&X[position \rightarrow P]\wedge\\&relativePosition(A, 0, D, 0, P) \wedge D > 0\end{aligned}$$

The user can then easily query the system without having to know internal details of the formal specification. Querying the system for all objects which are positioned on the left side of the object myCar can then be done with the following query:

$? - objectsLeftOf(myCar, X)$

11.2 Consistency checking

In this section we will illustrate by means of a small example how we can use the formalization to perform some consistency checks on the conceptual specifications. As we have seen for complex objects, an object can be part of another object. However, if somewhere in the conceptual specification an object a is modeled to be part of an object b but somewhere else in the conceptual specification b is stated to be a part of a then we have an inconsistency in our specification. The part-of relation is not symmetric.
We will now specify a predicate $partOfInconsistency(X, Y)$ as follows:

$$\begin{aligned} partOfInconsistency(X,Y) \leftarrow & X : concept \wedge Y : concept \wedge \\ & X[partOf \rightarrow Y] \wedge \\ & Y[partOf \rightarrow X] \end{aligned}$$

Now suppose that in the knowledge base we have the following facts:

$a[partOf \rightarrow b]$
$b[partOf \rightarrow a]$

Using a reasoning tool like Flora-2 we can now check if there is a part-of inconsistency in our knowledge base. This can be done by asking the following query:

$? - partOfInconsistency(D, E)$

When this query returns some values for D or E then we know immediately that there is a part-of inconsistency in our conceptual specification. The values of D and E will also indicate with which objects the inconsistency exists. In our case, a and b will be the values returned for D and E.

Part III

Implementation, Use Case and Conclusions

Chapter 12

Implementation

In chapter 4 we have informally described a set of high-level modeling concepts for complex objects for virtual environments. In chapters 7 to 10 we have unambiguously defined these modeling concepts by means of our formalization in F-logic.

In this chapter we will describe a number of prototype implementations. These implementations serve as a proof of concept for the feasibility of the modeling approach presented in this dissertation. This chapter is structured as follows: in section 12.1, a general overview of the implementations is given. Next, in section 12.2, we describe the diagram editor that has been implemented as an extension to Microsoft Visio [67]. This diagram editor supports the modeling of virtual environments using the VR-WISE graphical notations introduced in chapters 3 and 4. In section 12.3 we describe the Physics Generator Component (PGC) that has been build to support the specification of the desired mappings (see chapter 3) and to generate the actual VR application. The PGC has been developed specifically for the complex objects. In section 12.4, we illustrate how the PGC fits into the OntoWorld tool. The OntoWorld tool support the overall VR-WISE approach, including simple objects, complex objects and behavior.

12.1 General Overview

Figure 12.1 shows a general overview of the different prototype implementations that have been built. The diagram editor allows the user to draw the conceptual specifications using the graphical notation discussed in chapter 4. The diagram editor is described in more detail in section 12.2. Next we have developed the Physics Generator Component (PGC) (see section 12.3)which is a standalone application supporting the modeling of complex objects. This PGC allows to specify the mappings from the concepts and instances from the conceptual model to the virtual representation of it. Using these mappings it generates the complex objects inside the virtual environment. For the overall VR-WISE approach the OntoWorld tool has been built. The OntoWorld tool supports the complete VR-WISE approach, including simple objects, relations between objects such as spatial relations, complex objects and behavior. However, the PGC needs to be integrated with the OntoWorld tool in future. More details about this integration are given in section 12.4. The diagram editor communicates about the conceptual specifications with the OntoWorld tool by means of XML-files.

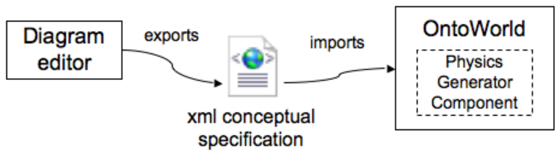


Figure 12.1: General overview of the implementation

12.2 Diagram Editor

The diagram editor provides a graphical interface for the specification phase in the VR-WISE approach. It allows the user to draw the conceptual specifications using the graphical notation developed for our modeling concepts. The graphical editor has been implemented as an extension to Microsoft Visio. Figure 12.2 shows the different components of the diagram editor. Note that the red components indicate the components of the diagram editor that have been extended or added in the context of the work presented in this dissertation.

The graphical user interface component accesses Microsoft Visio by means

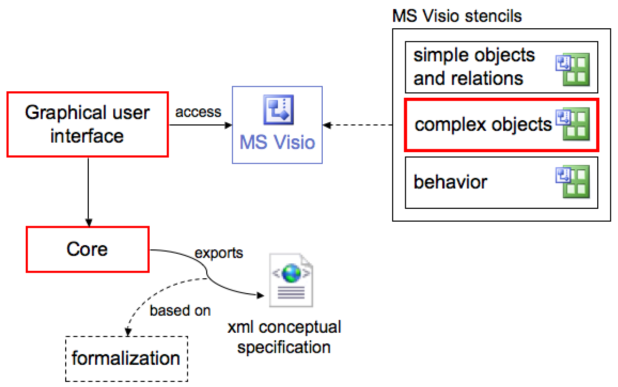


Figure 12.2: Diagram Editor overview

of an ActiveX control[1]. To use MS Visio, a number of Visio stencils have been created. Such a stencil contains graphical elements that can be used to create a particular diagram. For the graphical elements of the modeling concepts introduced in this dissertation, the *complex objects stencil* has been created. Figure 12.3 shows this Visio stencil.

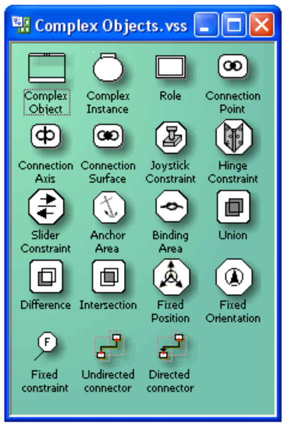
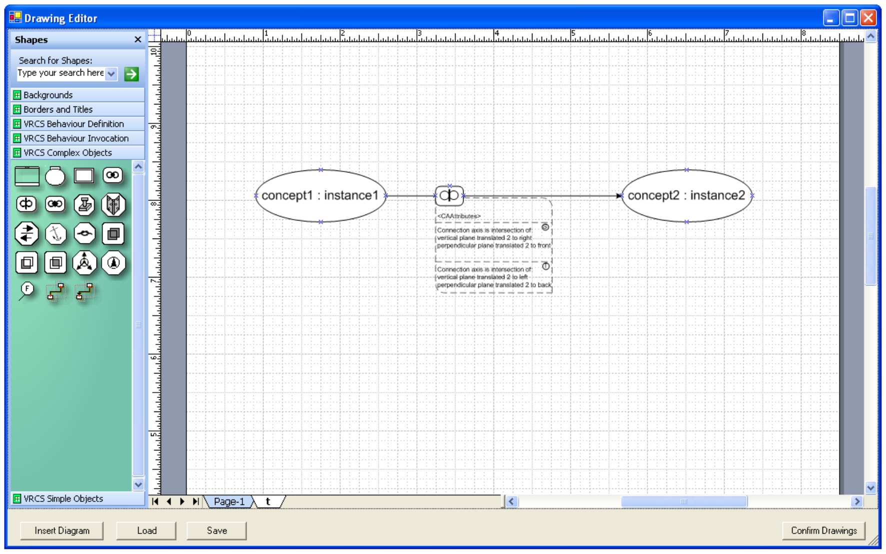


Figure 12.3: Visio stencil for complex object modeling concepts

The user can drag and drop the graphical elements from the stencils onto

[1]ActiveX is a Microsoft technology which allows the development of reusable object oriented software components.

the canvas and connect them using relations. As we have seen in chapter 4, some graphical elements have a simple and an extended notation (e.g., the connection relations). This is supported in the drawing editor. By double clicking on some graphical concept the user gets a graphical user interface that allows him to give the necessary details for the modeling concept. These details differ from modeling concept to modeling concept. Hence, for each modeling concept introduced in this dissertation, a new GUI component has been implemented. For example, when double clicking on a connection axis relation, a graphical interface is provided for specifying the connection axis. The expanding behavior of the modeling concepts together with the GUI for specifying the modeling concept details (attributes) has been implemented in C# [39] and is loaded as a COM object. Figure 12.4 shows a screenshot of the drawing editor.



<CAAttributes>

Connection axis is intersection of:
vertical plane translated 2 to right
perpendicular plane translated 2 to front

Connection axis is intersection of:
vertical plane translated 2 to left
perpendicular plane translated 2 to back

Figure 12.4: Screenshot from the graphical editor

Next, the core component of the diagram editor takes care of exporting the graphical conceptual models into an XML-format. For exporting the high-level modeling concepts introduced in chapter 4, this core component has also been extended. The high-level information from the graphical models is translated into a more low-level mathematical description. For example, a connection axis which is specified as the intersection of two planes in the conceptual model, is translated into the parametric equation for the connection axis before exporting it in the XML-format. Note that the calculation of the mathematical representations for the conceptual modeling concepts has been based on the mathematics as discussed in the formalization (see chapters 7 to 10). We illustrate this with a part of the XML-representation of a connection axis relation. As we can see, the connection axis element has a source and a target and it has the connection axis itself which is defined by means of the elements <a>, <b> and <c> representing the direction vector of the connection axis and the elements <x>, <y> and <z> representing a point on the connection axis.

```
<connection-axis>
  <caSource instanceRef="instance1">
  <caTarget instanceRef="instance2">
  <connectionAxis>
    <a>1</a>
    <b>1</b>
    <c>1</c>
    <x>1</x>
    <y>1</y>
    <z>1</z>
  </connectionAxis>
</connection-axis>
```

This type of information is easier to use when generating the virtual environment.

12.3 Physics Generator Component

As mentioned in the previous section, the diagram editor exports the conceptual specification in an XML-format. This XML-format can be imported in the OntoWorld tool. Once a conceptual specification is imported, the user needs to follow two steps. First the mapping step has to be performed; next the generation step can be executed. In order to support this process for complex objects, we have implemented the Physics Generator Component (PGC) as a standalone tool that allows demonstrating the feasibility of the conceptual modeling approach in the context of complex objects and shapes.

Mapping

During the mapping step the developer is asked to map the conceptual objects on an appropriate visual representation in the virtual environment.

Mappings can be defined for concepts as well as for instances. When a concept is mapped onto a visual representation, by default, all instances of this concept will be represented by this visual representation. However, to overwrite this default mapping inherited from its concept, an instance can also be mapped onto a visual representation. Figure 12.5 shows the object mapping window of the PGC. Note that this mapping step is the same as the mapping step performed in the general OntoWorld tool (see section 12.4). In figure 12.5 only some default VR primitives (box, cylinder, ...) are offered as the target for the mapping. However, the developer has the possibility to import other shapes. Importing a new shape happens by importing an X3D[2]-file. The content of such an X3D file looks as follows:

```
<ProtoDeclare name="shape name">
  <ProtoInterface>
    <field name="attibute name" type="type">         (1)
  </ProtoInterface>
  <ProtoBody>
    <Shape>
      <Appearance>
        ...
      </Appearance>
      <IndexedTriangleSet index= "0 2 1 2 3 5 ..."> (2)
        <Coordinate
         point=" 1 1 1, 1.5 1 2, 1 0 3, ...">       (3)
        </Coordinate>
      </IndexedTriangleSet>
    </Shape>
  </ProtoBody>
</ProtoDeclare>
```

So, X3D files that can be imported contain a *ProtoDeclare* node declaring the shape to import. Inside the *ProtoDeclare* node there is a *ProtoInterface* and a *ProtoBody* node. The *ProtoInterface* node contains all possible properties for the defined shape. Such a property definition is illustrated on line (1). It describes the name of the property and the type of the property. Examples of such attributes are the mass or the length of the shape. Next, the *ProtoBody* node contains the actual definition of the shape. The shape is specified by means of an *IndexedTriangleArray* node (see line (2)) representing a 3D shape composed of a collection of individual triangles. The *Coordinate* node

[2]http://www.web3d.org/x3d/

(line (3)) contains a number of vertices while the *index* field on line (2) specifies which vertices form a triangle. So, each three consecutive indices in the *index* field specify a triangle. If we look to the above example, then because of the first three indices 0 2 1 (on line (2)), the first triangle is formed by the coordinates (1,1,1), (1.5,1,2) and (1,0,3).

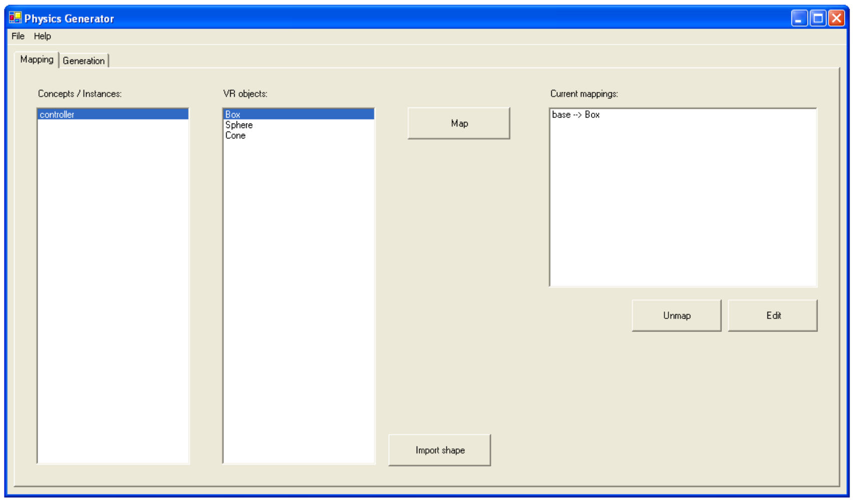


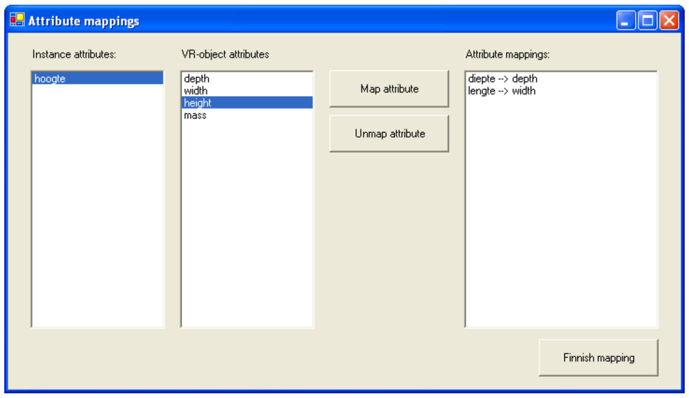
Figure 12.5: Concept/instance mapping window

For each object that is mapped onto a visual representation the developer may need to map the properties of that object onto properties of the visual representation. For example, the weight property of an instance car in the conceptual model can be mapped onto the mass property of its visual representation. The attribute mapping is shown in figure 12.6.

Generation

When all necessary mappings have been specified, the generation of the virtual environment can start. The generation of the PGC is illustrated in figure 12.7.

The PGC therefore uses the information available in the XML-specification of the conceptual model together with the information from the mapping step. The virtual environment is generated using the PhysX API, formerly known as the Novodex SDK. This is a high-performance C/C++ physics engine developed by Ageia. However, since the implementation of our tool



hoogte

depth
width
height
mass

diepte --> depth
lengte --> width

Figure 12.6: Attribute mapping window

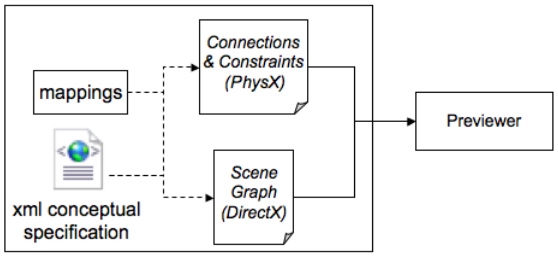

Connections & Constraints (PhysX)
mappings
Previewer
Scene Graph (DirectX)
xml conceptual specification

Figure 12.7: Code generation in the Physics Generator Component

is completely written in C# we use the Novodex wrapper[3]. The Novodex wrapper is a .NET wrapper for the PhysX API. The virtual environment generated by means of the PhysX API is using Direct3D[4]. The Direct3D API is part of Microsoft's DirectX. The outcome of this process is send to a previewer which shows the virtual environment to the developer. Figure 12.8 shows a screenshot from the GUI for a generated virtual environment.

So the PGC parses the conceptual specification in XML format. For each instance found in the conceptual specification the generator searches the appropriate mapping created during the mapping step. When the mapping is found, the generator creates the virtual object (using Direct3D). This virtual object can be a primitive VR object (such as a cylinder or sphere) or

[3]http://www.zelsnack.com/jason/JttZ/Novodex_NET_Wrapper
[4]http://www.microsoft.com/windows/directx/default.mspx

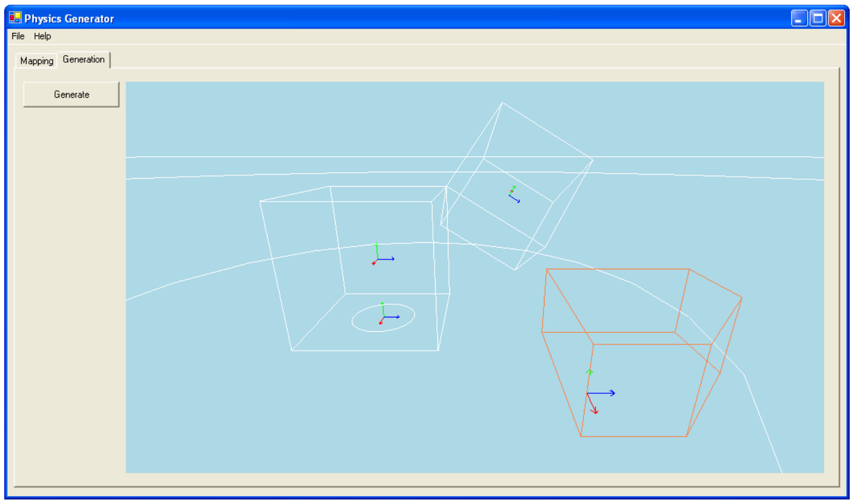


Figure 12.8: Generated virtual environment

it can be an imported shape specified in an X3D-file as we discussed above. When all objects have been created, the generator parses the connection relations and constraints from the conceptual model. For each of these relations or constraints between two instances, the appropriate joint (from the PhysX API) is created between the corresponding virtual objects. The PhysX API will then take care of satisfying all connections and constraints during the simulation.

12.4 Integration with OntoWorld

The OntoWorld tool supports the overall VR-WISE approach. It supports the creation of the conceptual models through the use of a GUI or by using the diagram editor described in section 12.2. It also allows specifying the mappings from the conceptual level to the implementation level. Note that this mapping process is the one described for the Physics Generator Component. Finally, it performs the generation of the actual virtual environment. This generation is using the XML-specification of the conceptual model. It is performed by the core component of the OntoWorld application. The generation process is shown in figure 12.9.

The core of the OntoWorld tool exists of several components. As we can see in figure 12.9, there is the statics generator component taking care

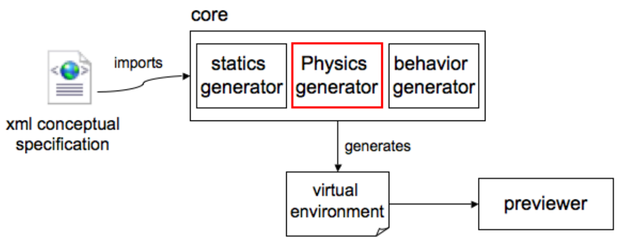


Figure 12.9: Generation process of the OntoWorld application

of the generation of the simple objects and the relations between objects such as the spatial relations. There is also a behavior generator component, which generates the behavior of objects. The generation part of the Physics Generator Component (see section 12.3) will need to be integrated into the physics generator component of the OntoWorld core. Then, the OntoWorld tool will take care of the mappings and will use the physics generator component for the generation of complex objects and shapes. Note that this integration is not yet realized. However, integrating the PGC into the OntoWorld tool will not impose a lot of problems since the integration has been taken into account when developing the PGC. Furthermore, the generated virtual environment can be viewed by the developer using the previewer.

Chapter 13

Case Study

In the previous chapter we have described our prototype implementation which serves as a proof of concept for the usability of the modeling concepts proposed in this dissertation and the feasibility of the VR-WISE approach in the context of complex object modeling. The aim of this chapter is to illustrate the modeling concepts for the conceptual specification of complex objects by means of an elaborated example. The outcome of this example can be fed to the proof-of-concept OntoWorld implementation which can then generate the corresponding virtual environment.

13.1 Subject

The subject of the case study is a virtual mechanical welding robot. The subject has been chosen in such a way that most of the modeling concepts defined in this dissertation are illustrated. The subject of this case study is illustrated in figure 13.1.
The welding robot consists of several parts. It has a *base* which can move along a *rail*. A mechanical arm stands on the base. The arm consists of two parts, a *lower arm* and an *upper arm*. On top of the *upper arm* there is a movable *welding head*.

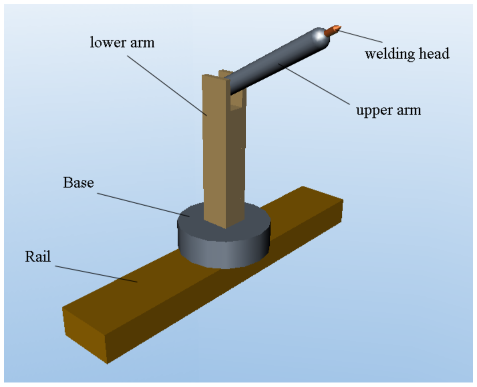


Figure 13.1: The virtual mechanical welding robot

13.2 Conceptual design

We will start the conceptual design by modeling the welding robot example on the domain specification level (see chapter 3). This level describes the concepts of the application domain and possible relations that hold between these concepts. We first identify and list the different concepts in the application domain together with their domain properties and default values. This information is given in table 13.1.
Before starting the actual modeling we will first describe how the different components are connected to each other. All these connections are illustrated in figure 13.2.

The *Base* of the robot is connected to the *Rail* over a connection axis in such a way that the *Base* and *Rail* are allowed to move a certain distance along the connection axis. Next, the *LowerArm* is connected to the *Base*

Table 13.1: Different concepts of the welding robot

Concept	**Attribute**	**Default value**
Rail	Length	6.0
	Height	0.4
	Width	1.0
	Weight	110
Base	Height	0.5
	Diameter	1.8
	Weight	90
LowerArm	Depth	0.5
	Height	2.5
	Width	0.5
	Weight	30
UpperArm	Length	2.0
	Radius	0.2
	Weight	15
WeldingHead	Length	0.3
	Size	0.1
	Weight	2

by means of a connection axis relation around which the components are allowed to rotate. The *UpperArm* is connected to the *LowerArm* by means of a hinge mechanism so that the *UpperArm* can move up and down. Finally, the *WeldingHead* is connected by means of a connection point to the *UpperArm*. The *WeldingHead* and *UpperArm* are further constrained so that the *WeldingHead* can only move as a joystick handle on the *UpperArm*.
After we identified all different concepts of our example together with the relations which connect them we can start the actual modeling of the welding robot. We will illustrate the modeling step by step. In the end, a complete conceptual model for the welding robot will be presented.

13.2.1 Base - Rail connection

The Base is connected to the Rail by means of a connection axis. For the *Rail* concept, the connection axis can be defined as the intersection of the perpendicular plane with the horizontal plane translated over half of the height of the *Rail* concept towards the top of the *Rail*. This is illustrated in

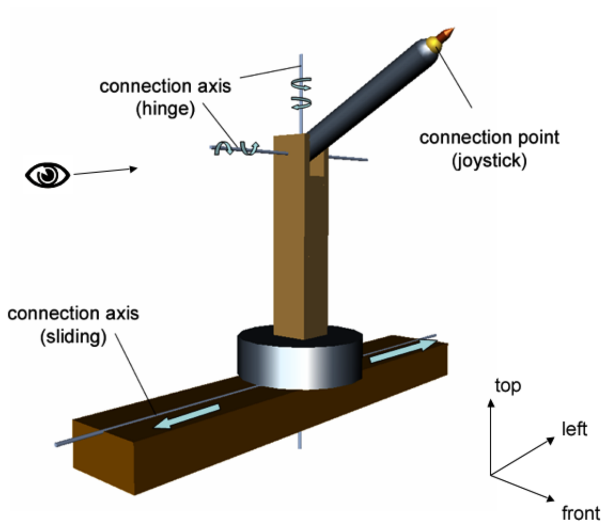


Figure 13.2: Connections between parts of the welding robot.

figure 13.3.

The connection axis on the *Base* concept is defined similarly. It can be specified as the intersection between the perpendicular plane and the horizontal plane translated to the bottom of the *Base* concept over half of the height of the *Base*. The conceptual specification of the connection axis relation between the *Rail* and *Base* concepts is given in figure 13.4.

A connection axis relation allows the connected concepts to rotate around or move along the connection axis. However, we only want to allow the Base to move along this connection axis. Therefore, as also shown in figure 13.4 we need to specify a slider constraint on top of the connection axis relation. This slider constraints allows the objects to move 2.5 units to the left and 2.5 units to the right starting from the initial positions of the connected objects with respect to each other.

13.2.2 LowerArm - Base connection

As illustrated in figure 13.2, the *LowerArm* is connected to the *Base* in such a way that the *LowerArm* can rotate infinitely around the connection axis. The connection axis lies exactly in the middle of the *LowerArm* and *Base* towards the top-to-bottom direction. Therefore, on both the *LowerArm* and the *Base* the axis can be defined as the intersection between the vertical

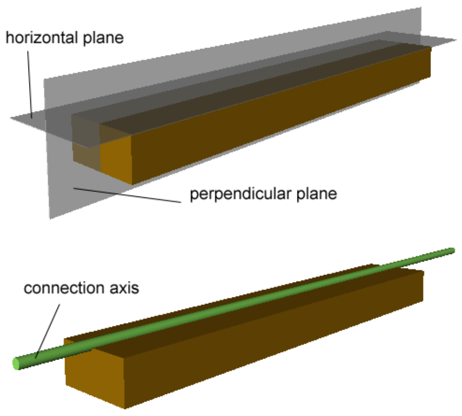


Figure 13.3: Ilustration of the specification of the connection axis on the Rail concept.

plane and the perpendicular plane.
However, if we would only specify the connection axis relation as explained above, the result of this connection relation would look as shown in figure 13.5. This is due to the fact that the objects are positioned along the connection axis in such a way that the translation points on both axes fall together. In order to get the desired result we need to translate the *LowerArm* along the axis towards the top direction. This is done by means of a translation point translation on the source side inside the connection axis specification.

Finally we need to constrain the motion of the *Base* and *LowerArm* in such a way that the objects may only rotate around the axis and not move along the axis. Since the *LowerArm* can rotate infinitely, it is sufficient to add a hinge constraint on top of the connection axis relation. No hinge limits need to be specified since no limits on a hinge constraint means that the objects may rotate infinitely. The conceptual specification for the connection between the *LowerArm* and the *Base* together with the hinge constraint is given in figure 13.6.

13.2.3 UpperArm - LowerArm connection

The UpperArm is connected to the LowerArm over a connection axis which serves as a hinge mechanism. Again, the connection axis is specified in a

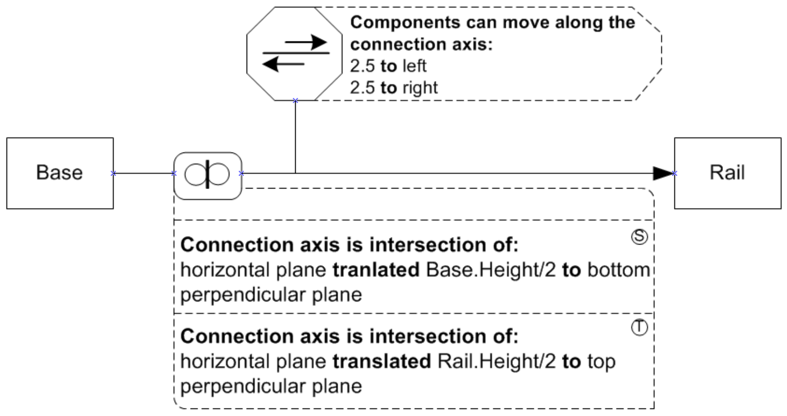

Components can move along the connection axis:
2.5 to left
2.5 to right
Base
Rail
Connection axis is intersection of:
horizontal plane tranlated Base.Height/2 to bottom
perpendicular plane
S
Connection axis is intersection of:
horizontal plane translated Rail.Height/2 to top
perpendicular plane
T

Figure 13.4: Conceptual specification for the Base to Rail connection.

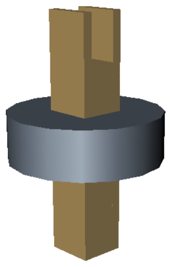


Figure 13.5: Connection between LowerArm and Base without translation point specification.

similar way as for the other connections we already discussed.
Next to the connection axis we also need to specify a hinge constraint that allows the UpperArm to move up and down with respect to the LowerArm. As explained in chapter 4, first we need to specify a viewpoint from which we look to the connection axis when specifying the limits of the hinge constraint. The viewpoint we will use here is from the right, as indicated in figure 13.2 by means of an eye icon.

From this viewpoint, moving the upper arm down is equal to a clockwise rotation of the upper arm with respect to the connection axis, while moving the upper arm up is equal to a counterclockwise rotation. We want the UpperArm to move up so that it becomes horizontal. This means from its

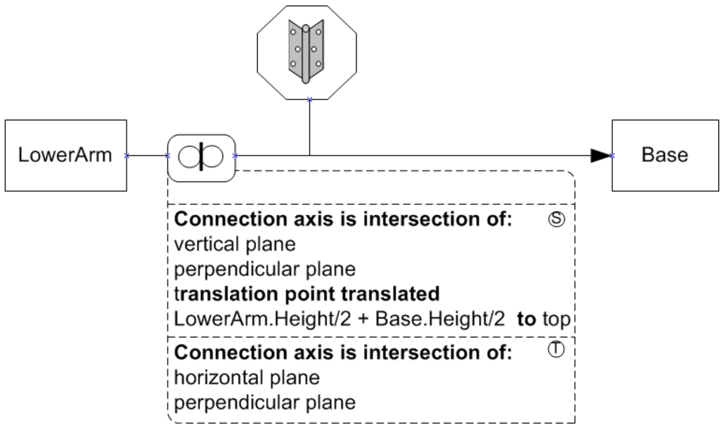

LowerArm
Base
Connection axis is intersection of:
vertical plane
perpendicular plane
translation point translated
LowerArm.Height/2 + Base.Height/2 to top
Connection axis is intersection of:
horizontal plane
perpendicular plane

Figure 13.6: Conceptual specification for the LowerArm to Base connection.

initial position (as shown in figure 13.2) the UpperArm can move 45 degrees counterclockwise around the connection axis. We also want the UpperArm to move down by rotating 90 degrees clockwise around the connection axis. The hinge constraint with the described limits is modeled in figure 13.7.

13.2.4 WeldingHead - UpperArm connection

The last connection we need to model is the connection between the WeldingHead and the UpperArm. First we need to model a connection point relation, next we need to constrain this relation by means of a joystick constraint. The conceptual specification of this connection is shown in figure 13.8.

A connection point on an object is specified relative to the position point. Since the position point of the upper arm is located exactly in the middle of the UpperArm, the connection point on the UpperArm is specified by translating the position of the UpperArm over a distance of half the length of the UpperArm towards the top of the UpperArm. This way we know that the connection point lies exactly on the top of the UpperArm. For the WeldingHead we translate the position point over a distance of half the length of the WeldingHead towards the bottom of the WeldingHead. This way the connection point lies on the bottom of the WeldingHead.

So far we have modeled the connection point relation. Next we need to constrain the allowed motion so that the objects are only allowed to rotate

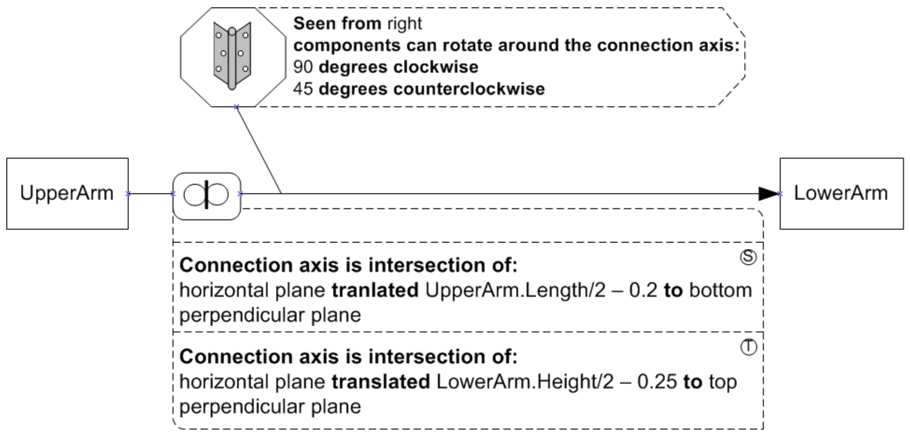

Seen from right
components can rotate around the connection axis:
90 degrees clockwise
45 degrees counterclockwise
UpperArm
LowerArm
S
Connection axis is intersection of:
horizontal plane tranlated UpperArm.Length/2 – 0.2 to bottom
perpendicular plane
T
Connection axis is intersection of:
horizontal plane translated LowerArm.Height/2 – 0.25 to top
perpendicular plane

Figure 13.7: Conceptual specification for the UpperArm to LowerArm connection.

around two perpendicular axes that go through the connection point. This is done by specifying a joystick constraint on top of the connection point relation. From the reference frame of the UpperArm we take the left-to-right axis and the front-to-back axis. Around each of these axes we want the WeldingHead to be able to rotate 45 degrees in each direction. Therefore, we constraint the rotation around both axes with an angle of 90 degrees (remember that a limit of 90 degrees means that the object may rotate 45 degrees clockwise and 45 degrees counterclockwise).

13.2.5 The overall conceptual model

The complete conceptual model for the complex concept *Welding_Robot* is given in figure 13.9. Note that the external orientation for both the UpperArm and the WeldingHead is changed by means of an orientation by angle relation. This is done so that the initial situation would look like in figure 13.1. Both concepts are rotated 45 degrees over the left-to-right axis of their reference frame.
Also note that the Rail concept has been chosen as reference concept of the complex concept. This means that the position and orientation of the complex concept is equal to the position and orientation respectively of the Rail concept.

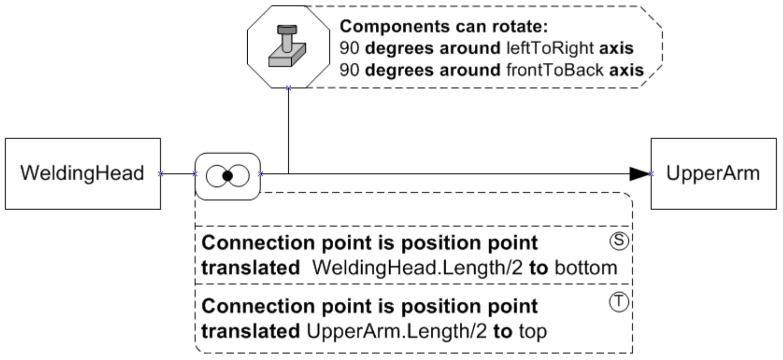

Components can rotate:
90 degrees around leftToRight axis
90 degrees around frontToBack axis
WeldingHead
UpperArm
Connection point is position point translated WeldingHead.Length/2 to bottom
S
Connection point is position point translated UpperArm.Length/2 to top
T

Figure 13.8: Conceptual specification for the WeldingHead to UpperArm connection.

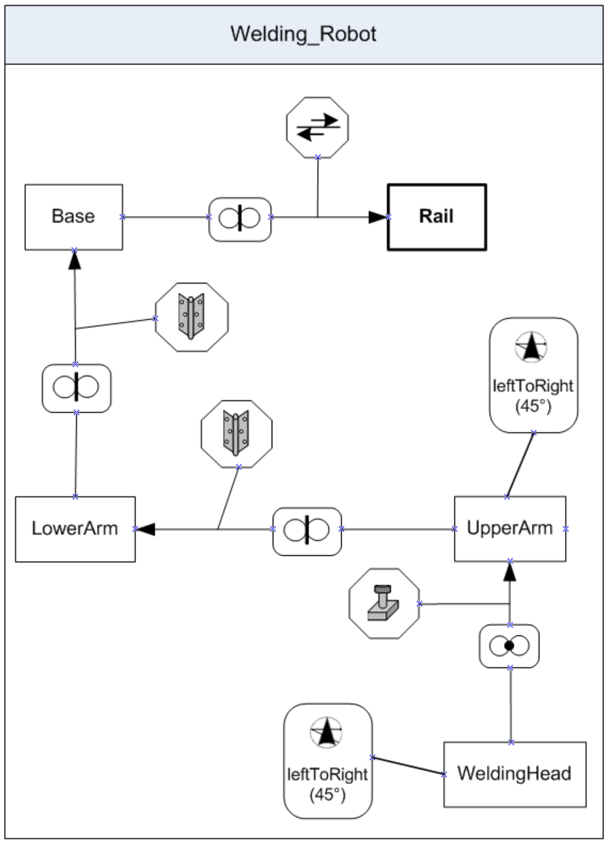


Figure 13.9: Complete conceptual specification for the welding robot

13.3 Generation of the virtual world

So far we have modeled the complex concept *Welding_Robot*. To generate the virtual environment we need to do two things. We need to give the concepts that are part of the *Welding_Robot* a default mapping towards their representation inside the virtual environment. Next we need to create an instance of the *Welding_Robot* which will be represented inside the virtual environment.

13.3.1 Mapping

From figure 13.1 it is easy to see that the Rail and Base concepts can be represented by VR primitives. A Rail can be represented as a box while the Base can be represented as a cylinder. The attribute mappings are trivial here. For example, the height, weight and $\frac{diameter}{2}$ of the Base are mapped on the height, mass and radius respectively of the cylinder.
However, the LowerArm, UpperArm and WeldingHead cannot be represented by VR primitives. Therefore we will map them on shapes specified inside an X3D-file (as an indexed triangle set). The representations of these concepts are illustrated in figure 13.10.

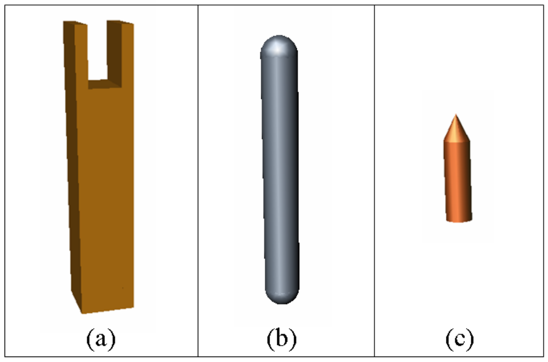


Figure 13.10: Representation in the virtual environment of (a) the LowerArm; (b) the UpperArm; and (c) the WeldingHead.

13.3.2 Instantiation

If we now want to create a virtual environment containing a welding robot, then we need to instantiate the *Welding_Robot* complex concept. The specification of an instance of this complex concept is given in figure 13.11. As can be seen from this figure, each concept which is part of the complex concept has been instantiated.

13.3.3 Generation

Now the conceptual specifications given in this chapter can be imported into our OntoWorld tool in order to generate the virtual environment. Since we didn't specify other mappings for the sub-instances of our robot instance, all parts will be represented by means of the default mappings associated with the concepts.

13.4 Limitations

In this chapter we have illustrated the modeling concepts for the conceptual specification of complex objects by means of an elaborated example. However, our approach also has some limitations. It will probably be difficult to use the VR-WISE approach to model for example detailed mechanical assemblies. This is due to the high-level of detail required for such types of objects. We can also think of other domains where the general VR-WISE approach can be difficult to apply because of domain specific concepts. However, in some situations these limitations may be solved by introducing a domain specific library of high-level modeling concepts.

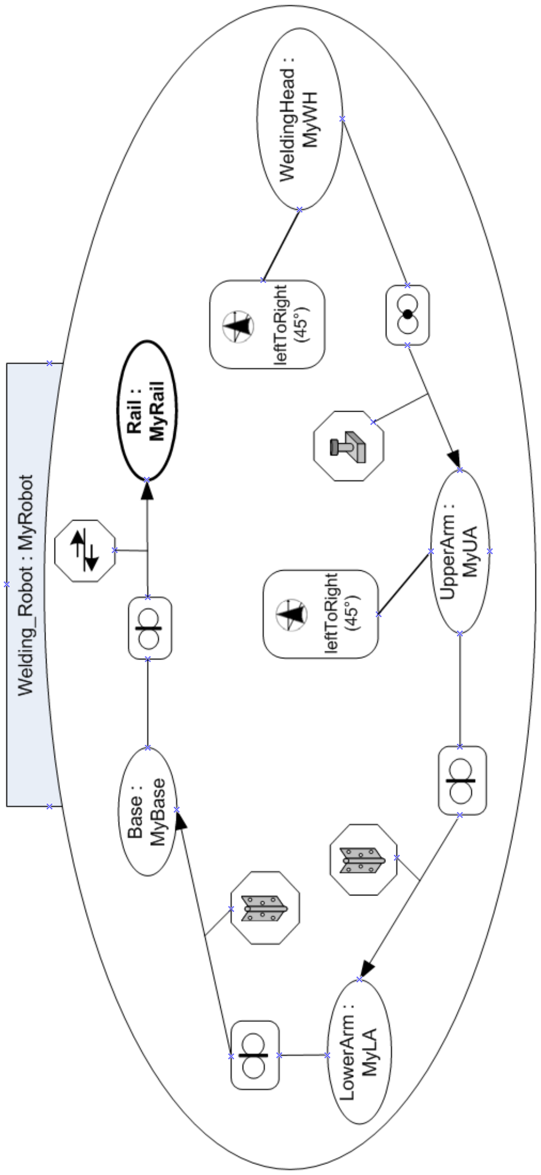


Figure 13.11: An example instance of the complex concept Welding_Robot.

Chapter 14

Conclusions

In this chapter we will summarize the work presented in this dissertation. We also take the opportunity to reflect on the contributions that have been delivered. Finally, we will list a number of limitations of the approach presented and we will discuss possible future work.

14.1 Summary

In this dissertation we have presented a new approach for modeling complex objects for Virtual Environments. The work presented in this dissertation fits into the VR-WISE framework which is a conceptual modeling approach for developing Virtual Environments. The VR-WISE approach introduces an explicit conceptual design phase in the development of a VR-application. During the conceptual design phase, high-level representations of objects inside the virtual environment, the relations that hold between these objects and the interaction between the objects and between the objects and the user, are specified. This allows domain experts to participate in the development of a VR-application. Also, the conceptual specifications can be used as a basis for discussions between the various stakeholders of a project. Finally, implementations can be build to transform these conceptual specifications into a working virtual environment.
In the first part of this dissertation, we have presented a number of high-level modeling concepts that can be used for specifying complex objects and complex shapes for virtual environments in the context of the VR-WISE approach. Several categories of modeling concepts can be distinguished. These are connection relations, connection constraints, constraints between connectionless groups of objects and CSG relations. Note that in most

toolkits the distinction between these categories is not made. If we look for example to the ODE [55] toolkit, all of the above modeling concepts are categorized as joints. For some connection relations in ODE the designer needs to create a joint and afterwards weaken the constraint limits on it. To our opinion it is more intuitive to have a number of basic connection relation which can be refined or constrained afterwards.
In the category of connection relations we have the connection point relation, the connection axis relation and the connection surface relation. These relations can be used to model a connection between two objects over a center of motion, over an axis of motion and over a surface of motion respectively. In the category of constraints on connections, we distinguish the hinge constraint, the slider constraint and the joystick constraint. The hinge constraint is always specified on top of a connection axis relation between two objects and constrains the connected objects in such a way that they can only rotate around the connection axis with respect to each other. Limits can be specified indicating how much the components may rotate around the connection axis. The slider constraint is specified on top of a connection axis relation between two objects. The slider constraint constrains the objects in such a way that they are only allowed to move along the connection axis. Also here, limits can be specified indicating how much the object may move along the connection axis. Finally, the joystick constraint is specified on top of a connection point relation. It constrains the motion of the connected objects in such a way that they may only rotate around two perpendicular axes through the connection point. Again, limits can be specified indicating how much the objects may rotate around each of these axes. Note that we have chosen to give the constraints a metaphor-based name. We believe that this is easier for non-technical persons to understand and remember their meaning.
Next, we have the category of constraints between connectionless groups of objects. In some situations it may be necessary to constrain for example the position or the orientation of objects while they are not physically connected. Therefore we incorporated a number of constraints to target these kinds of constraints between physically unconnected objects. These constraints are the fixed relative position constraint, the fixed relative orientation constraint and the positioning constraint. The fixed relative position constraint forces two objects to keep the same position relative to each other during their complete life-time while the fixed relative orientation constraint forces two objects to keep the same orientation relative to each other during their complete life-time. The positioning constraint specifies which objects can serve as a positioning base for which other objects. This constraint can

for example be used to indicate that a coffee cup can only be position on a saucer.
The last category of modeling concepts introduced in this dissertation is the set of CSG relations. For all the connections and constraints described above, the connected objects keep their own identity and can be manipulated individually in the virtual environment, as far as their connections and constraints allow. With CSG relations the connected objects loose their identity and are melted together to form one whole. Such a whole is called a complex shape. We introduced three CSG relations, which are the union relation, the intersection relation and the difference relation. For example, the union relation defines a new object as the union of two other objects. This means that the geometry of the union of two objects will consist of all points that are part of the geometry representing the first object and all points that are part of the geometry representing the second object. The definition of the intersection relation and the difference relation is similar.
Furthermore we introduced the modeling concepts *reference object* and *role*. An object that is part of a complex object can be the reference object of that complex object. This means that the position and orientation of the complex object is equal to the position and orientation of the reference object. Next, a role can be seen as a concept playing a certain role in the context in which it is used. The role modeling concept has been introduced to be able to reuse a concept that is needed several times (playing a different role) inside the specification of a complex concept.

In the second part of this dissertation we have given formal definitions for all the modeling concepts introduced in the first part of the dissertation, and also for existing modeling concepts in the VR-WISE approach. For this formalization we opted to use the logic-based formalism F-Logic because it fits the requirements we needed for the formalization language. The formalization given in this dissertation offers several benefits. Firstly, it unambiguously specifies the modeling concepts and this allows building unambiguous conceptual specifications. Secondly, the formalization offers a formal foundation that can be used for reasoning about the conceptual specifications. And thirdly, the formalization is independent from any implementation. Therefore, different implementations can be built based on the formal specification.
The formalization is comprised of actually three levels. The first level formally defines the syntax of the modeling concepts. The second and third level defines the semantics of the modeling concepts. On the second level, the formalization defines the semantics of a modeling concept on time zero

of the virtual environment. For example, a spatial relation specifies how two objects are positioned with respect to each other at time zero of the virtual environment. This level of semantics formalization is called the *initial semantics level*. The third level specifies the semantics of modeling concepts that also have semantics on times other than time zero in the virtual environment. For example, the connection axis relation not only takes care of correctly positioning two connected components along the connection axis at time zero of the virtual environment. During the rest of the simulation the components need to respect the connection axis relation in terms of positioning and orientation according to each other. This level of simulation semantics is therefore called the *simulation semantics level*.
In order to validate the modeling concepts proposed in this dissertation and to illustrate the feasability of the approach, a prototype implementation has been built. We built a diagram editor as an extension to Microsoft Visio which allows the user to draw the conceptual specifications using the graphical notation introduced in this dissertation. Next, we also built a tool that takes care of the complete process of translating the conceptual specifications into a working virtual environment. This tool is called OntoWorld.
Finally, we worked out a use case that serves to illustrate the use of the modeling concepts for complex objects inside the context of the VR-WISE approach. This is done by means of the elaborated example of a mechanical welding robot.

14.2 Contributions

In this section we will discuss the contributions and achievements resulting from this dissertation. First we will describe the contributions in relation with the problems and goals stated in the introduction (see chapter 1). Next, we will look to the contributions of the dissertation in the context of the overall VR-WISE research.

14.2.1 Contributions in relation to the problems and goals

- **The available tools for developing VR applications require a considerable knowledge about VR technology.** As explained in the introduction, the available tools for creating VR applications require a considerable knowledge about VR technology. The approach presented in this dissertation tries to avoid this by introducing a conceptual design phase into the development process of a VR application. When specifying concepts and instances, the designer can use domain

terminology instead of immediately having to specify the objects in terms of VR primitives. Concerning the specifications of connections and constraints, all connections and constraints can be described on a higher level of abstraction, reducing the need of VR knowledge. However, a basic mathematical knowledge and some spatial (three-dimensional) insight might be needed. For example when specifying a connection axis constraint the designer at least needs to know that the intersection between two planes defines an axis.
Note that VR knowledge might be needed when we come to the mapping. However, when the conceptual specifications need to be translated in terms of VR building blocks the designer can be assisted by a VR expert. At this stage of the development process a major part of the modeling has already been specified using the conceptual specifications. This way, the domain expert can be involved early in the design process, which lowers the chance of errors and misunderstandings early in the development process of a VR application.

- **The design phase in the development of a VR application is usually a very informal activity.** As stated in the introduction, few formal techniques in the context of VR exist to support the design phase effectively. This dissertation has made an attempt in providing such a technique by the introduction of a conceptual modeling phase. During this phase unambiguous conceptual specifications are built. By means of building the conceptual specification using high-level modeling concepts having a formal foundation, the design phase becomes a formal activity inside the design process of a VR application.

- **There does not exist a formal basis for discussing the design of a virtual environment between the different stakeholders of a project.** As stated in the introduction, often sketches and notes are used during the initial design. However, natural language and sketches are informal, ambiguous and often incomplete. As explained in chapter 2 conceptual models provide a communication platform between the various stakeholders in a project and help in the early detection of misunderstandings, errors and missing information. Therefore we believe that a VR-WISE conceptual model can serve as a basis for communication between designers, programmers and other stakeholders in a project. The conceptual specifications are easier to understand for the different stakeholders since they do not express the design in technical VR terms but on a higher level of abstraction using general knowledge and domain terminology. Furthermore, this dissertation also unambiguously specifies the modeling concepts by means of the proposed formalization. These modeling concepts are also complete in a sense that all information required for a certain connection or constraint is captured within one modeling concept. The unambiguousness and completeness of the conceptual specification may lead to fewer misunderstanding.

Furthermore, as stated in the goals in the introduction, we also wanted an approach that allows doing some intelligent reasoning. The approach presented in this dissertation offers this possibility through its formalization based on logic. This has been illustrated in chapter 11.

14.2.2 Contributions to the VR-WISE approach

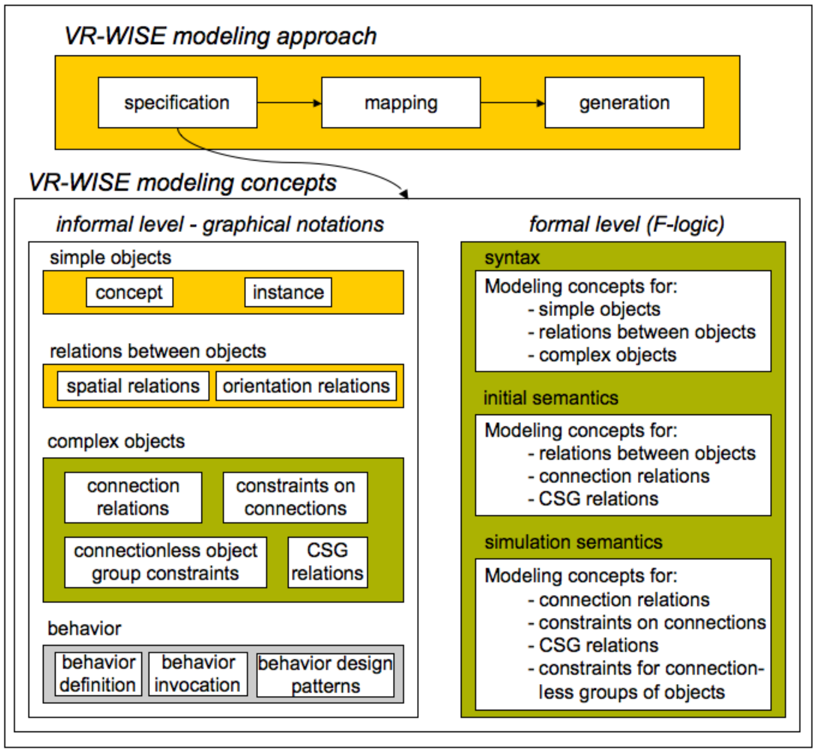


Figure 14.1: VR-WISE research results

Figure 14.1 gives an overview of the different research results inside the VR-WISE research project. They are divided into two parts. A first part consists of the VR-WISE approach itself which had been described in detail in chapter 3. The second part consists of the modeling concepts that can be used during the specification phase of the VR-WISE approach and their formal definition. The modeling concepts are grouped on the left side of figure 14.1. We will refer to them as the informal results because all the modeling concepts have been defined in an informal way. Also a graphical notation for each modeling concept has been provided. The right side of figure 14.1 concerns the formalization, which formally defines the modeling concepts. We will refer to them as the formal results.
The yellow blocks represent work preceding this dissertation and the achievement of several researchers at the WISE lab of the Vrije Universiteit Brussel. This includes the development of the overall process of the VR-WISE approach and the high-level modeling concepts for modeling simple objects, spatial relations, and orientation relations. Results of this work have been published in [62, 9, 10, 43].
The green blocks represent the work that has been performed specifically in the context of this dissertation. First of all, as informal results, this dissertation has introduced a set of high-level modeling concepts that can be used to model complex objects inside the context of the VR-WISE approach. Four categories of modeling concepts have been introduced which are connection relations, constraints on connection relations, modeling concepts for specifying complex shapes and finally a number of constraints that can be used for connectionless groups of objects.
The formal research results were completely obtained in the context of this dissertation. They define the modeling concepts for simple objects, relations between objects, and complex objects in an unambiguous way.
The grey block in figure 14.1 is work that has been performed parallel in the context of the PhD dissertation of B. Pellens [42]. This dissertation introduces high-level modeling concepts for modeling behavior for virtual environments.

Next, extensions have been made to the VR-WISE implementation. These have been described in chapter 12. The diagram editor has been extended with the modeling concepts presented in this dissertation. A proof-of-concept implementation, the Physics Generator Component, has been build to illustrate the approach in the context of complex object modeling. In future, most components of this Physics Generator Component will be integrated into the OntoWorld tool which supports the overall VR-WISE approach.

14.3 Limitations

As stated in the introduction, this dissertation does not claim to present a complete solution to all problems related to the development of virtual environments. In this section we will list the limitations that apply to the work presented in this dissertation:

- The first limitation is that the approach presented in this dissertation is only usable for modeling virtual environment up to a certain level of complexity. E.g., the approach presented is difficult to use for modeling detailed mechanical assemblies. This is due to the fact that these types of virtual objects require a very high level of detail and also because of the fact that domain specific concepts are needed. However, a layer can be built on top of our approach that pre-defines these necessary domain specific modeling concepts. Such an extension can be made for each domain. Also, our approach may be used for fast prototyping and for modeling the main lines of a virtual environment for such domains. Afterwards, the virtual environment generated from the conceptual specification may be refined by VR experts using other tools such as VR toolkits.

- A second limitation is that it is not yet possible to define a combination of connections and constraints between two components. This way, more powerful connections and constraints could be specified. As we have seen in the related work, for example with SimMechanics, joint composites are defined as a combination of some joint primitives. Such a mechanism can be powerful in defining new types of constraints. When looking to physics engines, we sometimes see they provide what is called a *carwheel constraint*. This is actually a combination of two hinge constraints comparable to the way a wheel is attached to a car. The problem however is that the motion allowed by one hinge constraint may be in contradiction to the motion allowed by the second hinge constraint. Indicating which motion allowed by which constraint has priority in the combination of constraints is easy to do inside a physics engine because of the access to all the underlying technical details of the constraints contained in the combination. In our approach however, there is no access to these low-level details.

Therefore, a new mechanism may need to be developed allowing to specify combinations of constraints and/or connections.

14.4 Future Work

- **Implementation of the formalization.** The formalization given in this dissertation unambiguously defines the modeling concepts inside the VR-WISE approach. As we have indicated in chapter 11, this formalization also allows to reason over the conceptual specification and to do some consistency checking. However, in order to make reasoning and consistency checking available, an implementation needs to be built.
  First of all, an extension of one of the available development environments for F-Logic is needed. Take for example Flora-2[1]. Although Flora-2 already has support for most of the mathematical operators used in our formalization, some operators may need to be included. However, this may not be a major problem since Flora-2 has been developed with extensibility in mind.
  Once such an extension has been implemented, the formalization can be placed inside a knowledge base. Note that we need several interconnected knowledge bases representing the different levels of our formalization. Once these knowledge bases have been defined they can be fed to the extended development environment. Conceptual specifications can then be translated into their corresponding F-logic representation and also added to the knowledge bases. Having all this information in F-Logic knowledge bases, we can use the extended Flora-2 to query the conceptual specifications and in such a way to do some reasoning and consistency checking.
  So far, the reasoning would happen by means of F-Logic queries. These queries are not very easy to build for people without knowledge about F-Logic. Therefore another idea for future work is to develop a high-level query language on top of the formalization implementation. Such a query language is comparable to SQL [37] in the domain of databases. The user can use it to interact with the conceptual specifications in a natural language-like way.

- **Dynamic conceptualizations.** As we have seen in chapter 2, the CODY Virtual Constructor [66, 33, 34] contains also , next to the

[1]http://flora.sourceforge.net/

static initial conceptual representation, a dynamically updated conceptual representation describing the current situation in the virtual environment. Such a dynamic conceptualization offers the advantage of reasoning in real time over the virtual environment. Based on the implementation of the formalization we have described above, it is only possible to reason over the initial conceptual specifications. However, in future work we could think of extending our current prototype implementation and the implementation of the formalization described above to be able to dynamically update our conceptual specifications. Therefore we would need to implement a mechanism so that changes in the generated virtual environment are reflected inside the conceptual specification. Such an extension would then allow us to query virtual environments modeled by means of our VR-WISE approach in real time.

- **Extension to the set of modeling concepts.** In this dissertation we introduced high-level modeling concepts for the most commonly used connections and constraints when modeling complex objects. However, this set of modeling concepts can be extended with other types of concepts. We only list a number of possibilities here.
  One possible extension is the introduction of high-level grouping patterns for objects. Often when we look to real world examples we see that objects are positioned in some pattern such as a line, a raster, a circle, ... We can think of examples like buildings and streets in Manhattan which are organized in some raster-like way or, the public of a football game organized in rows or lines. Therefore introducing grouping patterns may be interesting. This would allow the user to create automatically for example hundred instances of some domain concept and place them in a raster instead of having to model each instance separately and organize these instances in the desired way using spatial relations.
  Another possible extension would be to introduce a number of modeling concepts that allow automatic generation of instances of concepts in different ways. We could think of generating a number of instances of a concept which all follow the Gaussian distribution. This means that the values of some properties of the concept being instantiated a number of times are changed in such a way that these values form a Gaussian distribution. This way we could for example model a crowd by automatically instantiating a concept *human* a number of times following the Gaussian distribution. This means that not all

instances would look the same but that we could have small, large, thick, . . . humans. Note that maybe for some reason other statistical distributions can be implemented.
Next, when we look to related work we see that some of the approaches also contain a number of other modeling concepts. Take for example MotionWorks (see chapter 2). MotionWorks contains a number of so-called contact joints. This type of joints does not really describe a connection but rather a contact between two objects. An example of such a contact joint are two gearwheels of a watch which need to roll against each other. These may be modeling concepts to introduce later in the VR-WISE approach. However, which extensions are needed will follow from everyday use of the approach and will also differ from application domain to application domain.

Note that not only the modeling concepts for modeling complex objects may need to be extended but also other categories of modeling concepts in the VR-WISE approach. For example we might think of introducing n-ary spatial relationships. Such a spatial relation can for example be used to state that object A is positioned *between* object B and object C. The *between* spatial relation is an example of a ternary spatial relation.

Bibliography

[1] Extensible 3d (x3d) international standard. Technical report, Web 3D Consortium (Web3D), 2003.

[2] Motionworks 2004 tutorial, 2004.

[3] S. Abiteboul and S. Grumbach. Col: A logic-based language for complex objects. In *Proceedings of the International Conference on Extending Database Technology: Advances in Database Technology*, pages 271–293. Springer-Verlag London, 1988.

[4] Steven Aukstakalnis and David Blatner. *Silicon Mirage: The Art and Science of Virtual Reality*. Peachpit Press, 1992.

[5] Ronald T. Azuma. A survey of augmented reality. *Presence: Teleoperators and Virtual Environments*, 6(4):355–385, 1997.

[6] F. Baader, D. Calvanese, D. McGuinness, D. Nardi, and P. Patel-Schneider, editors. *The Description Logics Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

[7] Alistair P. Barros, Arthur H. M. ter Hofstede, and Henderik Alex Proper. Towards real-scale business transaction workflow modelling. In *Conference on Advanced Information Systems Engineering*, pages 437–450, 1997.

[8] Stefan Berner, Martin Glinz, and Stefan Joos. A classification of stereotypes for object-oriented modeling languages. In Robert France and Bernhard Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard. Second International Conference, Fort Collins, CO, USA, October 28-30. 1999, Proceedings*, volume 1723, pages 249–264. Springer, 1999.

[9] 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 the Workshop of Intelligent Computing*, pages 272–279, Mexico City, Mexico, 2004.

[10] 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, Volume I*, Madrid, Spain, 2004. IADIS PRESS.

[11] Dennis J. Bouvier. *Getting Started with the Java 3D API: A Tutorial for Beginners*, 2002.

[12] Grigore C. Burdea and Philippe Coiffet. *Virtual Reality Technology.* John Wiley & Sons, Inc., second edition, 2003.

[13] P. Chen. The entity-relationship model: Towards a unified view of data. *ACM Transactions on Database Systems*, 1(1):471–522, 1976.

[14] W. Chen and D.S. Warren. C-logic of complex objects. In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 369–378, Philadelphia, USA, 1989. ACM Press NY.

[15] Thomas Connolly, Carolyn Begg, and Anne Strachan. *Database systems: A practical approach to Design, Implementation and Management.* Addison Wesley, 1999.

[16] C. Cruz-Neira, D.J. Sandin, and T.A. DeFanti. Surround-screen projection-based virtual reality: The design and implementation of the cave. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, pages 135–142, 1993.

[17] J. de Bruijn, R. Lara, A. Polleres, and D. Fensel. Owl dl vs. owl flight: conceptual modeling and reasoning for the semantic web. In *Proceedings of the 14th international conference on World Wide Web*, pages 623–632, Chiba Japan, 2005. ACM Press London.

[18] Sharon J. Kemmerer (Ed). Step: The grand experience. NIST Special Publication 939, 1999.

[19] Clive Fencott. Towards a design methodology for virtual environments.

[20] Steven J. Fenves. A core product model for representing design information. Technical report NISTIR 6736, National Institute of Standards and Technology (NIST), 2001.

[21] Martin Fowler and Kendall Scott. *UML Distilled: a brief introduction to the standard object modeling language*. Addison-Wesley Professional, second edition, 1999.

[22] Kim G.J., Kang K. C., and Kim H. Software engineering of virtual worlds. In *Proceedings of the ACM symposium on Virtual Reality and Technology*, pages 131–138. ACM Press, 1998.

[23] Ontoprise GmbH. How to write f-logic programs covering ontobroker version 4.3, January 2006.

[24] Michael Gosele and Wolfgang Stuerzlinger. Semantic constraint for scene manipulation. In *Proceedings of the Spring Conference in Computer Graphics*, pages 140–146.

[25] T.R. Gruber. A translation approach to portable ontologies. *Knowledge acquisition*, 5(2):199–220, 1993.

[26] N. Guarino and P. Giaretta. *Ontologies and knowledge bases: towards a terminological clarification*, pages 25–32. ION Press, 1995.

[27] Terry Halpin. *Conceptual Schema and Relational Database Design*. WytLytPub, second edition, 1999.

[28] Terry Halpin. *Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design*. Morgan Kaufmann, first edition, 2001.

[29] Jed Hartman and Josie Wernecke. *The VRML 2.0 Handbook*. Addison Wesley, 1998.

[30] Cay Horstmann. *Object-oriented design and patterns*. John Wiley and Sons, 2003.

[31] S. Jayaram, H. Connacher, and K. Lyons. Virtual assembly using virtual reality techniques. *Journal of Computer-Aided Design*, 29(8), 1997.

[32] S. Jayaram, Y. Wang, U. Jayaram, K. Lyons, and P. Hart. A virtual assembly design environment. In *Proceedings of IEEE Virtual Reality Conference*, pages 172–180, Houston, Texas, USA, 1999.

[33] Bernhard Jung, Martin Hoffhenke, and Ipke Wachsmuth. Virtual assembly with construction kits. In *Proceedings of ASME Design Engineering Technical Conferences*, Sacramento, California, USA, 1997.

[34] Bernhard Jung and Ipke Wachsmuth. Integration of geometric and conceptual reasoning for interacting with virtual environments. In *Proceedings of the AAAI Spring Symposium on Multimodal Reasoning*, pages 22–27, Palo Alto, California, USA, 1998.

[35] M. Kifer and J. Wu. A logic for object-oriented programming (maier's o-logic revisited). In *Proceedings of the eighth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 379–393, Philadelphia, USA, 1989. ACM Press NY.

[36] Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.

[37] Kevin E. Kline. *SQL in a Nutshell.* O'Reily, 2 edition, 2004.

[38] Eric Lengyel. *Mathematics for 3D game programming and computer graphics.* Charles River Media Inc., 2nd edition, 2004.

[39] Jesse Liberty. *Programming C#.* O'Reilly, 2003.

[40] Wolfgang May. How to write f-logic programs in florid. http://dbis.informatik.uni-freiburg.de, accessed 6th of November 2006, 2000.

[41] Kelly L. Murdock. *3ds max 5 bible.* Wiley Publishing, 2003.

[42] B. Pellens. *A Conceptual Modelling Approach for Behaviour in Virtual Environments using a Graphical Notation and Generative Design Patterns.* PhD thesis, Vrije Universiteit Brussel, 2007.

[43] B. Pellens, W. Bille, O. De Troyer, and F. Kleinermann. Vr-wise: A conceptual modeling approach for virtual environments. In *CD-ROM Proceedings of the Methods and Tools for Virtual Reality workshop*, Gent, Belgium, 2005.

[44] B. Pellens, F. Kleinermann, W. Bille, and O. De Troyer. Overview of existing vr modeling concepts. Deliverable 1.1, VR-Demo Project (IWT 030248), 2004.

[45] B. Pellens, F. Kleinermann, O. De Troyer, and W. Bille. Model-based design of virtual environment behavior. In H. Zha et al., editor, *Proceedings of the 12th International Conference on Virtual Reality Systems and Multimedia*, pages 29–39, Xian, China, 2006. Springer-Verlag.

[46] B. Pellens, O. De Troyer, W. Bille, and F. Kleinermann. Conceptual modeling of object behavior in a virtual environment. In Xavier Fisher et al., editor, *Proceedings of Virtual Concept 2005*, pages 93–94, Biarritz, France, 2005. Springer-Verlag.

[47] 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 et al., editor, *Proceedings of Ontology Mining and Engineering and its use for Virtual Reality 2005*, pages 1215–1224, Agia Napa, Cyprus, 2005. Springer-Verlag.

[48] B. Pellens, O. De Troyer, F. Kleinermann, and W. Bille. Conceptual modeling of behavior in a virtual environment. *Special Issue of the International Journal of Product and Development*, pages xx–xx, 2006.

[49] S. Rachuri, Y-H. Han, S. Foufou, S. C. Feng, U. Roy, F. Wang, R.D. Sriram, and K. W. Lyons. A model for capturing product assembly information. *Journal of Computing and Information Science in Engineering*, 6(1):11–21, 2006.

[50] G. Robertson, M. Czerwinski, K. Larson, and M. van Dantzich. Immersion in desktop virtual reality. In *Proceedings of the 10th Annual Symposium on User Interfaces and Technology (UIST)*, pages 11–19, 1997.

[51] Ton Roosendaal and Stefano Selleri. *The official Blender 2.3 Guide: Free 3D creation suite for Modeling, Animation and rendering.* No Starch Press, 3 edition, 2005.

[52] Jarek R. Rossignac and Aristides A.G. Requicha. *Solid Modeling.* John Wiley, 1999.

[53] Jinseok Seo and Jounghyun Kim. Design for presence: A structured approach to virtual reality system design. *Presence*, 11(4):378–403, 2002.

[54] G. Smith and W. Stuerzlinger. Integration of constraints into a vr environment. In *Proceedings of the Virtual Reality International Conference 2001*, pages 103–110, Laval, France, 2001.

[55] Russell Smith. Open dynamics engine v5.0 user guide, 2004.

[56] Finnegan Southey and James G. Linders. Ossa — a conceptual modelling system for virtual realities. *Lecture Notes in Computer Science*, 2120:333–345, 2001.

[57] Wolfgang Stuerzlinger and Graham Smith. Efficient manipulation of object groups in virtual environments. In *Proceeding of the VR2002*, Orlando, Florida, 2002.

[58] Ivan E. Sutherland. A head-mounted three dimensional display. In *Proceedings of the Fall Joint Computer Conference*, pages 757–764, 1968.

[59] Vildan Tanriverdi and Robert J.K. Jacob. Vrid: A design model methodology for developing virtual reality interfaces. In *Proceedings of ACM Virtual Reality Software and Technology*, Alberta, Canada, 2001. ACM.

[60] A. H. M. ter Hofstede and Th.P. van der Weide. Fact orientation in complex object role modelling techniques. In T. A. Halpin and R. Meersman, editors, *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*, pages 45–59, Townsville, Australia, 1994.

[61] Alias Learning Tools. *The art of Maya: An introduction to 3D computer graphics.* Sybex, 3 edition, 2005.

[62] O. De Troyer, W. Bille, R. Romero, and P. Stuer. On generating virtual worlds from domain ontologies. In Tat-Seng Chua and Tosiyasu L. Kunii, editors, *Proceedings of the 9th International Conference on Multi-Media Modeling*, pages 279–294, Taipei, Taiwan, 2003.

[63] Olga De Troyer, Frederic Kleinermann, Haitem Mansouri, Bram Pellens, Wesley Bille, and Vladimir Fomenko. Developing semantic vr-shops for e-commerce. *Special issue of Virtual Reality in e-society for the Journal of Virtual Reality*, 2006.

[64] J.F.A.K. van Benthem, H.P. van Ditmarsch, J. Ketting, and W.P.M. Meyer-Viol. *Logica voor informatici.* Addison Wesley, 2 edition, 1993.

[65] John Vince. *Introduction to Virtual Reality.* Springer-Verlag, 2004.

[66] Ipke Wachsmuth and Bernhard Jung. Dynamic conceptualization in a mechanical-object assembly environment. *Artificial Intelligence Review*, 10(3-4):345–368, 1996.

[67] Mark H. Walker, Nanette J. Eaton, and Nanette Eaton. *Microsoft Visio 2003 Inside Out*. Microsoft Press, 2003.

[68] Guizhen Yang, Michael Kifer, Chang Zhao, and Vishal Chowdhary. *Flora-2: User's Manual*, 2005.

[69] Thomas G. Zimmerman, Jaron Lanier, Chuck Blanchard, Steve Bryson, and Young Harvill. A hand gesture interface device. *ACM SIGCHI Bulletin*, 17(SI):189–192, 1986.