Defining the Semantics of Conceptual Modeling Concepts for 3D Complex Objects in Virtual Reality

Olga De Troyer, Wesley Bille, and Frederic Kleinermann

Vrije Universiteit Brussel
Research Group WISE, Pleinlaan 2
1050 Brussel
Belgium
olga.detroyer@vub.ac.be,
wesley.bille@skynet.be,
frederic.kleinermann@vub.ac.be
http://wise.vub.ac.be/

Abstract. Virtual Reality (VR) allows creating interactive three-dimen sional computer worlds in which objects have a sense of spatial and physical presence and can be manipulated by the user as such. Different software tools have been developed to build virtual worlds. However, most tools require considerable background knowledge about VR and the virtual world needs to be expressed in low-level VR primitives. This is one of the reasons why developing a virtual world is complex, time-consuming and expensive. Introducing a conceptual design phase in the development process will reduce the complexity and provides an abstraction layer to hide the VR implementation details. However, virtual worlds contain features not present in classical software. Therefore, new modeling concepts, currently not available in classical conceptual modeling languages, such as ORM or UML, are required. Next to introducing these new modeling concepts, it is also necessary to define their semantics to ensure unambiguousness and to allow code generation. In this paper, we introduce conceptual modeling concepts to specify complex connected 3D objects. Their semantics are defined using F-logic, a full-fledged logic following the object-oriented paradigm. F-logic will allow applying reasoners to check the consistency of the specifications and to investigate properties before the application is actually built.

Keywords: Virtual Reality, F-logic, semantics, complex objects, conceptual modeling, formal specifications, VR-WISE.

1 Introduction

Virtual Reality (VR) is a technology that allows creating interactive three-dimensional (3D) computer worlds (virtual worlds, also called Virtual Environments or VE's) in which objects have a sense of spatial and physical presence

S. Spaccapietra, L. Delcambre (Eds.): Journal on Data Semantics XIV, LNCS 5880, pp. 1–36, 2009.
© Springer-Verlag Berlin Heidelberg 2009

and can be manipulated by the user as such. VR has gained a lot of popularity during the last decennia due to games and applications such as Second Life [1]. A lot of different software tools have been developed which allow building VE's. However, most tools require considerable background knowledge about VR technology. The development of a VE directly starts at the level of the implementation. A developer needs to specify the VE using the specialized vocabulary of the VR implementation language or framework used. Therefore, when creating a VE the objects from the problem domain have to be translated into VR building blocks (such as textures, shapes, sensors, interpolators, etc.), which requires quite some expertise. This makes the gap between the application domain and the level at which the VE needs to be specified very large, and makes the translation from the concepts in the application domain into implementation concepts a very difficult issue. This is one of the reasons why developing a VE is complex, time-consuming and expensive.

In different domains such as Databases, Information Systems and Software Engineering, conceptual modeling has been used with success to support and improve the development process. The term conceptual modeling denotes the activity of building a model of the application domain in terms of concepts that are familiar to the application domain experts and free from any implementation details. Conceptual modeling has less been used in 3D modeling and VR. However, like for these other domains, introducing a conceptual design phase in the development process of a VR application could be useful to improve and support the development of VE's. It will reduce the complexity and can provide an abstraction layer that hides the specific VR jargon used. In this way, no special VR knowledge will be needed for making the conceptual design of a VE and also non-technical people (like the customer or the end-user) can be involved in the development process. A conceptual model 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 VE, earlier detection of design flaws is possible. All this could help in realizing more VR applications in a shorter time.

However, conceptual modeling for VR poses a lot of challenges as VE's involve a number of aspects, not present in classical software or information systems. VE's are 3D worlds composed of 2D and 3D objects and often deal with 3D complex objects for which the way the parts are connected will influence the way the complex objects can behave (i.e. *connected complex objects*). Furthermore, to realize dynamic and realistic worlds, objects may need complex (physical) behaviors. This requires new modeling concepts, currently not available in classical conceptual modeling languages, such as ORM [3] [4] or UML [2]. Next to introducing new modeling concepts, it is also necessary to define their semantics. Defining the semantics of the modeling concepts will allow for unambiguous specifications. Unambiguousness is important from two perspectives. Firstly, if the semantics of the modeling concepts are clear, the models created with these modeling concepts will be unambiguous and there will be no discussion between different stakeholders about their meaning. Secondly, unambiguousness is also

needed from the perspective of the code generation; otherwise it will not be possible to automatically generate code.

In this paper, we introduce conceptual modeling concepts to specify connected complex 3D objects, as well as their semantics. These modeling concepts are part of the VR-WISE approach. VR-WISE (Virtual Reality - With Intuitive Specifications Enabled) [5] [6] [7] [8] [9] is a conceptual model-based approach for VR-application development. The semantics of the modeling concepts presented here are defined formally using F-logic, a full-fledged logic following the object-oriented paradigm.

The rest of this paper is structured as follows. Section 2 will provide the background. It includes an introduction to VR (section 2.1), VR-WISE (section 2.2), and F-logic (section 2.3). In section 2.1, we will briefly discuss the different components of a VR application as well as how VR applications are developed these days. In section 2.2, we will discuss the limitations of current conceptual modeling techniques with respect to the modeling of VR, and briefly introduce VR-WISE, the conceptual modeling approach developed for VR. In section 3, we discus the conceptual modeling of complex connected 3D objects and introduce the related conceptual modeling concepts in an informal way. Next, in section 4, the semantics of these modeling concepts will be defined. In section 5 we will discus related work. Section 6 concludes the paper and points out further work.

2 Background

In this section, we introduce some background material. In section 2.1, we briefly discuss the different components of a VR application as well as how VR applications are developed these days. In section 2.2, we discuss the limitations of current conceptual modeling techniques with respect to the modeling of VR, and briefly introduce VR-WISE, the conceptual modeling approach developed for VR. In section 2.3, F-logic is introduced.

2.1 VR

There are many definitions of Virtual Reality (VR) [10] [11]. For the context of this research, VR is defined as a three-dimensional computer representation of a space in which users can move their viewpoints freely in real time. We therefore consider the following cases being VR: 3D multi-user chats (Active Worlds [12], first person 3D videogames (Quake [13]) and Unreal tournament ([14]), and 3D virtual spaces on the Web (such as those created with VRML [15], and X3D [16]). In this section, we will define the main components of a VR application (VE) and then briefly review how a VE is developed today.

Main components of a VR application. A VE is made of different components [10], which can be summarized as:

1) **The scene and the objects.** The scene corresponds to the environment (world) in which the objects are located. It contains lights, viewpoints and cameras. Furthermore, it has also some properties that apply to all the objects located inside the VE, e.g., gravity. The objects have a visual representation with color and material properties. They have a size, a position, and an orientation.
2) **Behaviors** .The objects may have behaviors. For instance, they can move, rotate, change size and so on.
3) **Interaction.** The user must be able to interact with the VE and its objects. For instance, a user can pick up some objects or he can drag an object. This may be achieved by means of a regular mouse and keyboard or through special hardware such as a 3D mouse or data gloves [10].
4) **Communication.** Nowadays, more and more VE are also collaborative environments in which remote users can interact with each other. To achieve this, network communication is important.
5) **Sound.** VR applications also involve sound. Some research has been done over the last ten years in order to simulate sound in a VE.

Developing a VE. The developing of the different components of a VE is not an easy task and during the last fifteen years, a number of software tools have been created to ease the developer's task. These tools can be classified into authoring tools and software programming libraries.

Authoring tools. Authoring tools allow the developer to model the static scene (objects and the scene) without having to program. Nevertheless, they assume that the developer has some knowledge of VR and some programming skills to program behaviors using scripting languages. Different authoring tools may use different scripting languages. The most popular authoring tools are 3D Studio Max [20], Maya [21], MilkShape 3D [22], various modelers such as AC3D [23] and Blender [24]. If the developer is developing for a certain file format, he needs to pay attention to the file formats supported by the authoring tool.

Programming Libraries. With programming libraries a complete VE can be programmed from scratch. Among the existing libraries, there is Performer [25], Java3D [17], X3D toolkit written in C++ [26] or Xj3D [27] written on top of Java3D. To use such a library, good knowledge of programming and a good knowledge of VR and computer graphics are required. It is also possible to use a player that, at run-time, interprets a 3D format and build the VE. VRML [15] and X3D [16] are 3D formats that can be interpreted by special players through a Web browser. Examples of such players are the Octaga player [18] and the Flux player [19].

We will not discus here how the other components of a VE (behavior, interaction, etc.) are development these days; it is not directly relevant for the rest of the paper. In any case, we can conclude that although there are quite a number of tools to help a developer to build the scene of a VE, until now, the general problem with these tools and formats is that they are made for VR specialists or at least for people having programming skills and background in computer graphics or VR. In addition, there is also no well-accepted development method for VR.

Most of the time, a VR-expert meets the customer (often the application domain expert) and tries to understand the customer's requirements and the domain for which the VE is going to be built. After a number of discussions, some sketches are made and some scenarios are specified. Then, the VR-expert(s) start to implement. In other words, the requirements are almost directly translated into an implementation. This way of working usually result into several iterations before the result reaches an acceptable level of satisfaction for the customer. Therefore, the development process is time consuming, complex and expensive.

2.2 VR-WISE

Introducing a conceptual design phase in the development process of a VR application can help the VR community in several ways. As conceptual modeling will introduce a mechanism to abstract from implementation details, it will reduce the complexity of developing a VE and it avoids that people need a lot of specific VR knowledge for such a conceptual design phase. Therefore, also non-technical people (like the customer or the end-user) can be involved 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 VE, earlier detection of design flaws is possible. And finally, if the conceptual models describing the VR system are powerful enough, it may be possible to generate the system (or at least large parts of it) automatically.

Several general-purpose conceptual modeling languages exist. Well-know languages are UML [2], ER [28] and ORM [3] [4]. ER and ORM were designed to facilitate database design. Their main purpose is to support the data modeling of the application domain and to conceal the more technical aspects associated with databases. UML is broader and provides a set of notations that facilitates the development of a complete software project. To a certain extend, UML, ORM and ER could be used to model the static structure of a VR application (i.e., the scene and the objects), however, all are lacking modeling concepts in terms of expressiveness towards VR modeling. For example, they do not have built-in modeling concepts for specifying the position and orientation of objects in the scene or for modeling connected objects using different types of connections. Although, it is possible to model these issues using the existing modeling primitives, this would be tedious. E.g., each time the modeler would need a particular connection he would have to model it explicitly, resulting in a lot of "redundancy" and waste of time. In addition, the models constructed in this way would not be powerful enough to use them for code generation because the necessary semantics for concepts like connections would be lacking. Furthermore, neither ORM nor ER provides support for modeling behavior.

It could be possible to extend these general-purpose modeling languages with new modeling concepts to enable VR modeling. However, another approach regarding this problem is the creation of a Domain Specific Modeling Language. We have opted for this last approach because we want to have a modeling language, as well as a modeling approach, that is easy and intuitive to use also for non VR-experts. The modeling approach taken by a general-purpose language

such as UML is very close to the way software is implemented by means of OO programming languages. The use of (an extended) UML would also force the use of its modeling paradigm. It is our opinion, that for certain aspects of VR, this would not be the best solution. Therefore, we have developed VR-WISE, a domain specific modeling approach for the development of VE's.

VR-WISE includes a conceptual specification phase. During this conceptual phase, conceptual specifications (so-called conceptual models) are created. Such a conceptual specification is a high-level description of the VE, the objects inside the environment, the relations that hold between these objects and how these objects behave and interact with each other and with the user. These conceptual specifications must be free from any implementation details. Therefore, the approach offers a set of high-level modeling concepts (i.e. a modeling language) for building these conceptual specifications. As indicated, we require that these modeling concepts are very intuitive, so that they can be used, or at least be understood, by different stakeholders. This means that the vocabulary used, should be familiar to most of its users. Because we also opted for a model-driven approach, the expressive power of the different modeling concepts must be sufficient to allow code generation from the models.

The conceptual specification consists of two levels since the approach follows to some degree the object-oriented (OO) paradigm. The first level is the *domain specification* and describes the *concepts* of the application domain needed for the VE (comparable to object types or classes in OO design methods), as well as possible relations between these concepts. In the overall example that we will use, we will consider a VE containing virtual mechanical robots. This VE could be used to illustrate the working of those robots. For such an application, the domain specification could contain concepts such as Robot, WeldingRobot, LiftRobot, Controller, WorkPiece, Box, and relations such as "a Robot is driven-by a Controller". Concepts may have *properties* (attributes). Next to properties that may influence the visualization of the concepts (such as height, color, and material) also non-visual properties, like the cost and the constructor of a robot, can be specified. At this conceptual level, we only consider properties that are conceptual relevant. Properties like shape and texture are not necessarily conceptual relevant and may depend on how the object will be visualized in the actual VE. The visualization of the objects is considered in a later phase in the VR-WISE approach. For a VE, behavior is also an important feature. However, the focus of this paper is on modeling concept for complex objects, therefore we will not elaborate on behavior. Details on modeling concepts for behavior can be found in [29] [30] [31] [32] [33] [34].

The second level of the conceptual specification is the *world specification*. The world specification contains the conceptual description of the actual VE to be built. This specification is created by instantiating the concepts given in the domain specification. These *instances* actually represent the objects that will populate the VE. In the robot example, there can be multiple Robot-instances and multiple WorkPiece-instances. Behaviors specified at the domain level can be assigned to objects.

Objects in a VE have a *position* and an *orientation* in the scene (defined in a three-dimensional coordinate system). Although it is possible to specify the position of the instances in a scene by means of exact coordinates and the orientation by means of angles, we also provide a more intuitive way to do this (more suitable for non-technical persons). If you want to explain to somebody how the robot room should look like, you will not do this in term of coordinates. Instead you will say that: "Two welding robots are in front of each other at a distance of one meter. A lift robot is left of each welding robot, and a box is placed on the platform of each lift robot". In such an explanation, spatial relations are used to describe the space. As spatial relations are also used in daily life, they provide a good intuitive way to specify a scene. Therefore, they are available as modeling concepts. Note that although the use of spatial relations may be less exact than coordinates, they are exact enough for a lot of applications. A *spatial relation* specifies the position of an object relative to some other object in terms of a direction and a distance. The following directions may be used: *left, right, front, back, top*, and *bottom.* These directions may be combined. However, not all combinations make sense. For example, the combined direction *left top* makes sense, but *left right* doesn't. Spatial relations can be used in the domain specification as well as in the world specification. In the domain specification, the spatial relations are used between concepts and specify default positions for the instances of a concept. The spatial relations currently supported are the most common ones. It is also possible to consider others, like for instance an "inside" relation. Currently, "inside" can be modeled by considering the object in which another object has to be placed as a new scene.

In a similar way, *orientation relations* can be used to specify the orientation of objects. For example, the *orientation by side relation* is used to specify the orientation of an object relative to another object. It specifies which side of an object is oriented towards which side of another object. E.g., one can specify that the front of the instance WeldingRobot1 is oriented towards the backside of the instance WeldingRobot2.

As common for conceptual languages, the conceptual modeling concepts of VR-WISE also have a graphical notation.

2.3 F-Logic

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 is a frame-based language; the central modeling primitives are classes with properties (attributes). These attributes can be used to store primitive values or to relate classes to other classes. Subclasses are supported. In this section, we provide a brief introduction to F-logic in order to make the paper self-contained. This introduction is based on [35] and [36] to which we refer the interested reader for more details.

Class Signatures. A class signature specifies names of properties and the methods of the class. To specify an attribute definition $\Rightarrow$ is used, $\Rightarrow\!\!\!\!\!\Rightarrow$ is used to express a multi-valued attribute.

The following statement gives the class signature for the class $professor$:

professor[*publications* $\Rightarrow\!\!\!\Rightarrow$ *article;*
dep $\Rightarrow$ *department;*
highestDegree $\Rightarrow$ *string;*
highestDegree $\bullet\!\!\rightarrow$ *"phd"*]

publications $\Rightarrow\!\!\!\Rightarrow$ *article* states that publications is a multi-valued property. *highestDegree* $\Rightarrow$ *string* states that highestDegree is a property of type string, and *highestDegree* $\bullet\!\!\rightarrow$ *"phd"* states that it is an inheritable property, which has the effect that each member-object of the class professor inherits this property and its value. E.g., a member *bill* will have the property *highestDegree* with value *phd* by inheritance. An inheritable property is inherited by a subclass. The inheritable property remains inheritable in this subclass while an inheritable property inherited by a member of the class becomes non inheritable.

Class Membership. In F-Logic we use ":" to represent class membership.

mary : professor
cs : department

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 a great deal of uniformity.

Method Signatures and Deductive Rules. Next to properties, classes can have methods. Consider the following class:

professor[publications $\Rightarrow\!\!\!\Rightarrow$ *article;*
dep $\Rightarrow$ *department;*
highestDegree $\Rightarrow$ *string;*
highestDegree $\bullet\!\!\rightarrow$ *phd;*
boss $\Rightarrow$ *professor]*

The property *boss* is actually a method without arguments. In F-Logic, there is no essential difference between methods and properties. The method *boss* takes no arguments as input and gives an object of type professor as output. The following statement is the deductive rule defining the method *boss* for objects of the class $professor$.

$$
\begin{aligned}
P[boss \rightarrow B] \leftarrow\ & P : professor \wedge \\
& D : departement \wedge \\
& P[dep \rightarrow D[head \rightarrow B : professor]]
\end{aligned}
$$

The previous statement 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.

It is also possible to create methods that take one or more 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. The following statement 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 $\Rightarrow\!\!\!\!\!\!\Rightarrow$ *article]*

subclassess. "::" is used to represent the subclass relationship. The following statements denote that $employee$ is a subclass of $person$ and that $professor$ is a subclass of $employee$.

employee :: person
professor :: employee

Predicate. In F-logic, predicate symbols can be used in the same way as in predicate logic, for example:

promotorOf(mary, bill)

Queries. Queries can be considered as a special kind of rules, i.e. rules with an empty head. The following query requests all members of the class $professor$ working at the department cs:

?- X : professor $\wedge$ *X[dep* $\rightarrow$ *cs]*

3 Conceptual Modeling Concepts for Connected Complex 3D Objects

In section 2, we have presented an overview of VE's and the VR-WISE approach. As already indicated, complex 3D objects, and more in particular connected complex objects, are important in the context of VE's. In our robot example, a welding robot is a complex connected object, composed of a rail, a base, a lower arm, an upper arm, and a welding head (see figure 1). To enable the specification of such concepts, we need dedicated conceptual modeling concepts.

Let's first start with giving an informal definition of a "complex object" in the context of a VE.

Complex objects are built from other simple and/or complex objects. They are composed by connecting two or more simple and/or complex objects. The connected objects are called components. All components keep their own identity and can be manipulated individually. However, manipulating a component may have an impact on the other components of the complex object. The impact depends on the type of connections used.

Looking to this description, the following issues seem to be important. (1) Complex objects are composed of other objects (components), (2) components

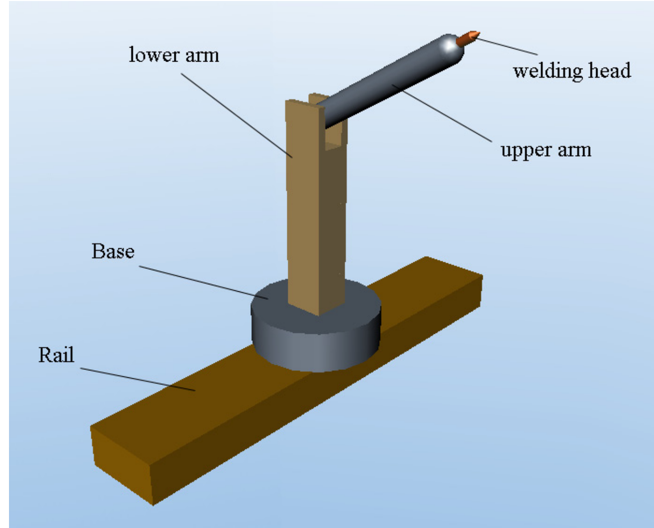

Fig. 1. An illustration of a welding robot

are connected by means of connections and there exist different types of connections, and (3) the motion of a component may be restricted by the connection type used. Let's explain this last issue. Normally an object has six degrees of freedom, three translational and three rotational degrees of freedom. The translational degrees of freedom are translations along the three axes of the coordinate system used while the three rotational degrees of freedom are the rotations around these three axes. 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. Here, we will discuss three possible connection types, namely over a center of motion, over an axis of motion and over a surface of motion. Other types of connections are possible and can be defined in a similar way. In section 6 on future work, some examples of other types of connections are given. The connection types that we consider here are abstractions from specific connection types usually available in 3D modeling tools (such as ODE [37], PhysX [38], MotionWork [39]). In our conceptual modeling approach, we specify the type of connection between components by means of the so-called *connection relations*. Note that these connection relations can also be used to specify the connection between objects without the purpose of defining a complex object. For example, we can define a connection between a boat and the water surface, but we don't want to consider the boat and the water surface as one complex object.

To support the connection over a center of motion, respectively over an axis of motion and over a surface of motion, we have defined the *connection point relation*, respectively the *connection axis* relation and the *connection surface* relation. We describe them in more detail in the following sections using the

welding robot as an example. However, connection types on their own are not sufficient to come to realistic behaviors. To further restrict the behaviors of the connected components, constraints will be used. For instance when you want to state that the components cannot be moved relative to each other. More details on constraints are given in section 3.4. Note that we use the term "components" to refer to the objects involved in a connection relation. However, this does not imply that the use of a connection relation implies defining a complex object. The connection relations are only used to specify connections between objects. Also note that we don't consider here connecting and disconnecting objects at runtime. This is part of the behavior specification, which is outside the scope of this paper.

3.1 Connection Point Relation

A first way of connecting two objects to each other is over a center of motion. In the real world we can find examples of objects connected over a center of motion, e.g., the shoulder of the human body connecting the arm to the torso. In the welding robot example, the welding head is connected to the upper arm over a center of motion to allow the welding head to rotate in different directions. A center of motion means that there is somewhere a point in both components that needs to coincide during the complete lifetime of the connection. We call this point the *connection point*. Connecting two objects over a center of motion removes all three translational degrees of freedom of the components with respect to each other. Specifying a connection point relation implies specifying the connection point for both components. This can be done by means of exact coordinates, however we are looking for a method that is more intuitive for the layman. Therefore, the position of the connection point is specified relative to the *position point* of the object. This is a (default) point in the object that is used to specify the position of the object in the VE, i.e. when an object is positioned at the coordinates (x,y,z) this means that the position point of the object is at position (x,y,z).

Figure 2 shows our graphical notation of the connection point relation. Boxes represent the components; connection relations are represented by a rounded rectangle connecting the two components by means of an arrow. The icon inside the rounded rectangle denotes the type of connection, here a connection point relation. The arrow indicates which component is the source and which is the target. The source should be connected to the target, i.e. in figure 2 component A is connected to component B. Hence, figure 2 can be read as "A is connected by means of a connection point to B". Note that this is not necessarily the same as connecting B to A using the same connection relation. When the composition should be performed if the two composing objects already have a position (e.g., at runtime), it may be necessary to reposition one of the objects. The convention is that the source will be repositioned.

Note that the graphical notation given in figure 2 does not specify the actual connection points. This is done by means of a simple markup language and using the expanded graphical notation (see figure 3). Allowing to hide or to omit

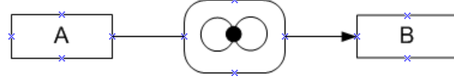

Fig. 2. Graphical notation of connection point relation

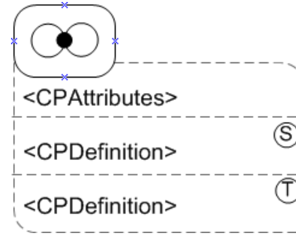

Fig. 3. Extended Graphical notation for a connection point relation

the details of the connections relations is useful as abstraction mechanism in different phases of the design and for different stakeholders. The expanded area has three sub areas. The top area is used to specifying general properties of the connection. Currently, the only possible attribute for the connection point relation is the stiffness. Current values for the stiffness are 'soft', 'medium' or 'hard'. The second and third areas hold the definition of the connection point for the source component respectively for the target component.

The position of a connection point is specified relative to the position point of the component. This is done in terms of zero or more translations of this position point. If no translations are given the connection points coincides with the position point of the object. A translation is specified by a distance and a direction. The distance is expressed by an arithmetic expression and a unit (if no unit is given the default unit will be used). Note that inside the arithmetic expression, object properties can be used. They allow referring to properties of the components, e.g., its width. The direction is given by means of keywords: *left, right, front, back, top* or *bottom.* These directions may be combined. However, not all combinations make sense. A combined direction exists of minimal two and maximal three simple directions. For example, we may use the combined direction *left top*, but *left right* is meaningless. In this way 'translated 2 cm to left' specifies that the connection point is defined as a translation of the position point 2 cm towards the left side. Please note that the position point itself is not changed. The syntax is as follows:

<CPAttributes> ::= [*<stiffness>*]
<stiffness> ::= 'connection stiffness is' <stiffnessType>
<stiffnessType> ::= 'soft' | *'medium'* | *'hard'*
<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 >) |
<arithmetic expression> <operator> <arithmetic expression>

*<operator> ::= + | - | * |*

Figure 4 gives an example of a connection point relation. It is used to connect the welding head to the upper arm of the welding robot. The welding head is the source and the upper arm is the target. Since, the position point of the upper arm is defined in the middle of the upper arm (by default the position point is the centre of the bounding box of the object), the connection point for the upper arm is specified as a translation of the position point over half of the length of the upper arm towards the top. In this way the connection point is exactly on the top of the upper arm. Similar, for the welding head, the connection point (which should be at the bottom of the welding head) is specified as a translation of the position point over half of the length of the welding head towards the bottom.

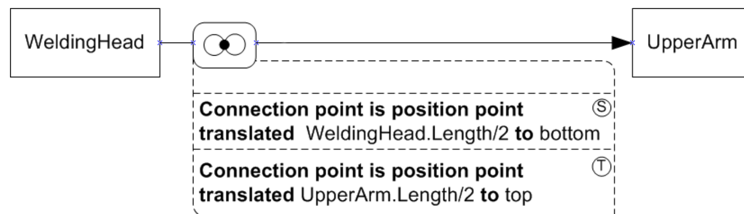

Fig. 4. Example connection point relation

3.2 The Connection Axis Relation

A second way to connect two components is over an axis of motion. A lot of examples of this connection type can be found in the real world. For example: a wheel that turns around an axis, a door connected to a wall, the slider of an old-fashioned typing machine. Actually, an axis of motion means that there is an axis that restricts the displacements of the components 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 removes four degrees of freedom leaving only one translational and one rotational degree of freedom.

To specify a connection axis relation between two components, we actually have to specify the connection axis for each of the two components. These two axes need to coincide during the complete lifetime of the connection. Looking for an easy way to specify these axes, we decided to allow a designer to specify an axis as the intersection between two planes. Therefore, three planes through each object are predefined. These are the *horizontal plane*, *the vertical plane* and *the perpendicular plane*. These planes are illustrated in figure 5.

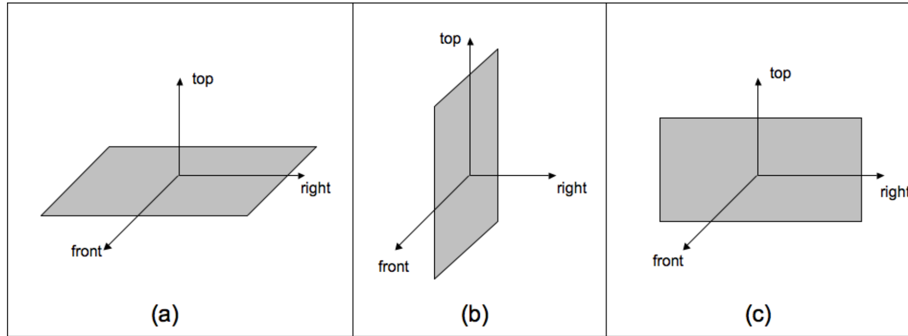

Fig. 5. (a) the horizontal plane; (b) the vertical plane; (c) the perpendicular plane

A connection axis is defined as the intersection between two of these planes. To allow more flexibility, the predefined planes can also be translated or rotated. 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. This is done by specifying for each component a point on its connection axis. These points should coincide. By default this connection point is the orthogonal projection of the position point of the component onto the connection axis. However, our approach also allows the designer to change this default by translating this default point along the connection axis.

The graphical representation of the connection axis relation is similar to that of the connection point relation (see figure 6).

Also in this case, the graphical notation is expandable (see figure 7). The second and third areas are now used for the definition of the connection axis for the source, respectively the target. The syntax is as follows:

<CAAttributes> ::= [<stiffness>]
<stiffness> ::= 'connection stiffness is' <stiffnessType>
<stiffnessType> ::= 'soft' | 'medium' | 'hard'
<CADefinition> ::= 'connection axis is intersection of: '
<planeDefinition>

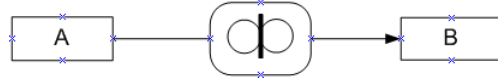

Fig. 6. Graphical represenation of the connection axis relation

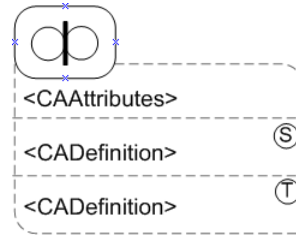

Fig. 7. Expanded graphical representation of the connection axis relation

<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*>

As an example we show how the base of the welding robot is connected to the rail by means of a connection axis relation to allow the base and the rail to move along this axis. The specification is given in figure 8. For the rail, the connection axis is specified as the intersection of the perpendicular plane with the horizontal plane translated over half of the height of the rail towards the top of the rail. This is illustrated in figure 9. The connection axis on the base is defined as the intersection of the perpendicular plane with the horizontal plane translated to the bottom of the base over half of the height of the base.

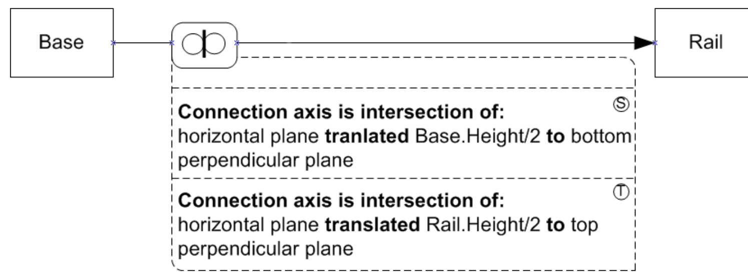

Fig. 8. Example connection axis relation

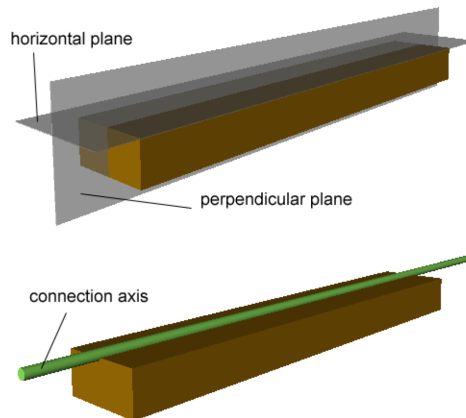

Fig. 9. Illustration of the definition of a connection axis

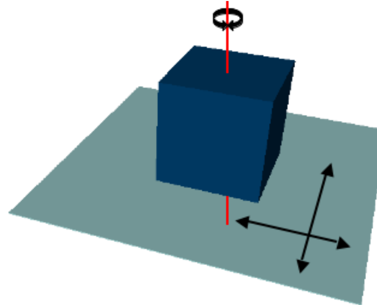

Fig. 10. Degrees of freedom for the connection surface relation

3.3 The Connection Surface Relation

The last way to connect two components to each other that we want to discuss here is over a surface of motion. A real world example of this type of connection is a boat able of floating over a water surface. A surface of motion means that there is a surface that allows the components to move along the directions of this surface. This connection type removes three degrees of freedom. The only degrees of freedom left 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 10. The surface of motion is called the *connection surface*.

To specify a connection surface relation we actually need to specify the connection surface for each of the components. The connection surfaces of both components need to coincide during the complete lifetime of the connection. To specify these connection surfaces, again we apply the three predefined planes (the horizontal plane, the vertical plane, and the perpendicular plane). For each of the components, the designer selects an initial plane to work with. This plane can be translated and rotated. Similar as for the connection axis relation we also need a connection point to specify the initial position of both components on the connection surface. By default, this point will be the orthogonal projection of the position point of the component on the corresponding connection surface. Also for the connection surface relation, this point can be translated to specify other positions. The graphical representation of the connection surface relation is similar as that of the other connection relations (see figure 11).

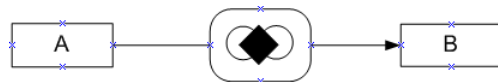

Fig. 11. Graphical representation of the connection surface relation

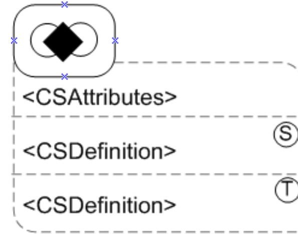

Fig. 12. Extended graphical representation of the connection surface relation

The expanded graphical notation has again three areas: one for specifying the properties of the relation, one area for the specification of the connection surface for the source and one for target. The extended graphical notation is illustrated in figure 12.

The syntax is as follows:

<CSDefinition> ::= 'connection surface is: '
<planeDefinition> [*<CSConnectionPoint>*]

<CSConnectionPoint> ::= 'connection point is positioning point '
<distance> 'to ' <direction>
[*'and ' <distance> 'to ' <direction>*]

In figure 13 a connection surface relation is used to specify that a box placed on the platform of a lift robot should only be able to move over this platform. The connection surface for the box is defined as the default horizontal plane of the box translated towards the bottom of the box; the connection surface for the platform is also the horizontal plane (but this time it is the horizontal plane of the platform itself because the platform is a plane itself and therefore it will coincide with its horizontal plane).

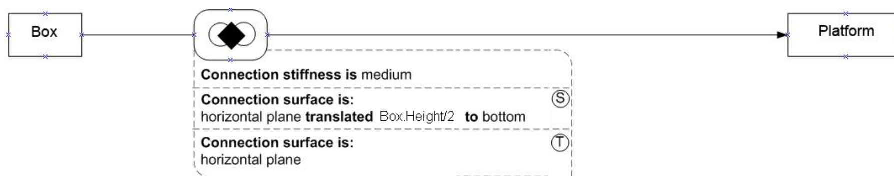

Fig. 13. Connection Surface for lift robot with a platform

3.4 Constraints on Connections

So far we are able to specify connection relations between components. As discussed, these relations impose a limitation on the degrees of freedom of the components with respect to each other. However, this is not always sufficient to come to realistic behaviors. For example, by means of the connection axis relation used to connect a base to its rail, it is still possible to rotate the base and the rail around the connection axis. This is not what we want. We would like to be able to specify that the base should only be able to move along its connection axis. Instead of defining this as yet another special kind of connection relation, we have opted to specify these kinds of restrictions by means of constraints that can be specified on top of the connection relations. For our base-example, a constraint can be attached to the connection axis relation stating that the base may only move over a given distance along its connection axis. A number of constraints are predefined, e.g., the *hinge constraint*, the *slider constraint* and *the joystick constraint*. The names of the constraints are metaphor-based which should make it easier for non-technical persons to understand and remember their meaning. For example, the restriction of the base motion can be expressed by a slider constraint.

A slider constraint can be defined on top of a connection axis relation to restrict the motion to a move along the connection axis. Furthermore, the move can be limited by indicating how much the components may move along the connection axis. Figure 14 illustrates the specification of a slider constraint for the welding robot. The constraint is defined on top of the connection axis relation that connects the base to the rail. The base can move 2,5 units to the left and to the right.

A hinge constraint is also specified on top of a connection axis constraint and restricts the motion to a rotation around the connection axis. It is also possible to indicate limits for this movement. The joystick constraint restricts

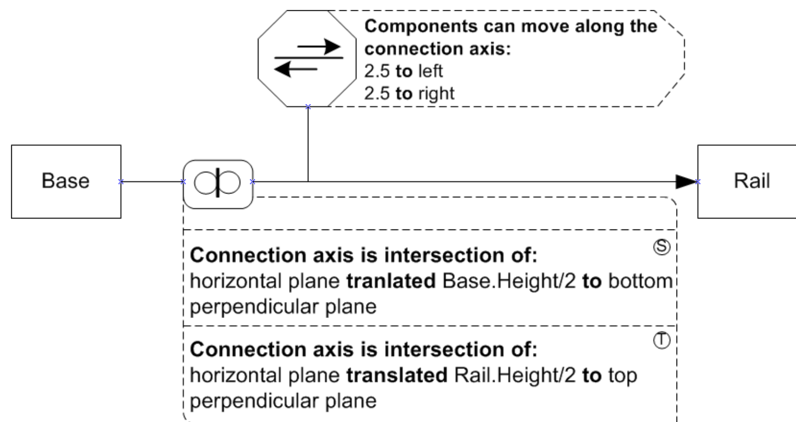

Fig. 14. An example of a slider constraint

the motion of two components connected by means of a connection point relation to a rotation around two perpendicular axes through the connection point. A joystick constraint can also have limits indicating how much the components may rotate around the axes in the clockwise and in the counterclockwise direction. More information on these constraints can be found in [5].

4 Formal Specification of the Modeling Concepts

In this section we will illustrate how the semantics of the modeling concepts introduced in the previous section can be defined rigorously. Due to space limitations, it is not possible to give the complete formalization of the modeling concepts introduced in this paper. For the complete formalization we refer to [5]. We will focus on the principles used and give a representative number of formalizations.

4.1 Principles

To define the semantics of a modeling concept, we will express what its use means for the actual VE. E.g., if we state that two objects are connected by means of a connection axis relation then the semantics of this relation should define the implication for the position and orientation of the components in the VE. To be able to do this, we need a formalism that is able to deal with the instance level. However, because the instance level is defined through the concept level, we also have to deal with the concept level. This means that defining the semantics of the modeling concepts (i.e. the meta-level) requires access to the concept level and to the instance level (see figure 15).

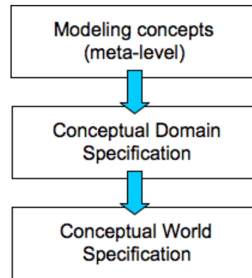

Fig. 15. The three levels involved in the formalization

In F-logic, classes can be treated as objects, which allows the meta modeling that we need here.

4.2 Formalization of the Basic Modeling Concepts

In this section, we formally define some of the basic modeling concepts used in the VR-WISE approach.

Point. A point in a three dimensional space can be given by means of an x, y and z coordinate. Using the F-logic formalism, a point is defined as a class with x, y and z properties of type float.

$point[x \Rightarrow float;$
$y \Rightarrow float;$
$z \Rightarrow float]$

Orientation. Each object in a VE has an orientation. In the VR-WISE approach, each object specified in the world specifications will have a default orientation. This default orientation is illustrated in figure 16. The default orientation of an object can be changed by rotating the object around on or more of the axes of the reference frame used. In order to be able of expressing the orientation of objects, we define the class orientation with properties *frontAngle*, *rightAngle* 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.

orientation[frontAngle $\bullet\!\!\to$ *0;*
rightAngle $\bullet\!\!\to$ *0;*
topAngle $\bullet\!\!\to$ *0]*

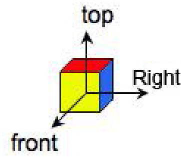

Fig. 16. Default orientation on an object

Line. We have formally defined 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 [$x_0 \Rightarrow$ *float;*
$y_0 \Rightarrow$ *float;*
$z_0 \Rightarrow$ *float;*
$a \Rightarrow$ *float;*
$b \Rightarrow$ *float;*
$c \Rightarrow$ *float]*

4.3 Formalization of Concept and Complex Concept

Concept. A concept is the main modeling concept in the domain specification. We have defined a concept in F-logic as a class. So domain concepts are represented as classes in F-logic. Each class representing a domain concept needs to be defined as a subclass of the predefined class *concept*. The class *concept* is defined as follows:

concept[position ⇒ point;
internalOrientation ⇒ orientation;
externalOrientation ⇒ orientation]

The properties *position*, *internalOrientation*, and *externalOrientation* are methods. The instances of the concepts are the actual objects in the VE. Objects in a VE have a position and an orientation. In VR-WISE we make use of an internal orientation and an external orientation (explained further on). The corresponding methods in the class concept are inheritable properties and therefore each instance of any subclass will inherit them. These methods can therefore be used to return the position, respectively internal and external orientation of an object. The definitions of these methods are rather elaborated (and therefore omitted), as they need to take into account whether the object is a component of some other object and how its position and orientation has been specified. I.e., as explained earlier, objects can be positioned using exact coordinates but also relative to other objects using spatial relations. Therefore, the position can either be exact or relative. When the designer specifies the position by means of coordinates (a point), the exact position is known and there is no need to calculate the position. However, when the position is given by means of some spatial relations or connection relations, only relative positions are given and the actual coordinates need to be calculated. Also the orientation can be exact (given by means of angles) or relative (given by means of orientation relations).

Because of the use of orientation relations in VR-WISE, each object has been given an internal and an external orientation. The *internal orientation* of an object is used to specify which side of the object is defined as the front, back, left, right, top and bottom side. The *external orientation* of an object is used to specify the orientation of the object in the VE. This works as follows.

By default each object has an own reference frame. This is the coordinate system local to the object where the origin of the coordinate system is the origin of the object (which is also the position point). Figure 17(a) illustrates the default internal orientation for an object. By default, the top/bottom, front/back, and right/left side of an object are defined as in Figure 17(a). It is possible to change this default by means of a rotation of the local reference frame of the object around some of the axes of the global reference frame. (The global reference frame refers to the coordinate system of the VE itself. The origin of the global reference frame is the origin of the VE.) Figure 17(b) illustrates a non-default definition of top/bottom, front/back, and right/left. This is done by specifying an internal orientation which is a 45 degrees counterclockwise rotation of the default internal orientation around the front direction. Note that the object

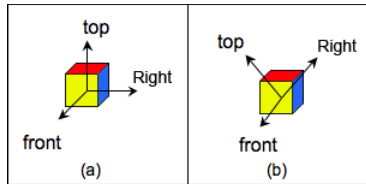

Fig. 17. (a)default internal orientation; (b) internal orientation 45 degrees counterclockwise around front axis

itself is not rotated. Actually, changing the internal orientation only redefined the left-right and top-bottom sides of the object.

An object also has a default external orientation. Figure 18(a) illustrates the default external orientation for an object. The external orientation of an object can be used to change the orientation of the object in the VE. This is done by a rotation of the object around some of the axes of the object's local reference frame, which will result in a rotation of the object itself. This is illustrated in figure 18(b) where the default external orientation is changed by means of a rotation of 45 degrees counterclockwise around the front axis. As you can see, the complete object has been rotated and as such its orientation in the VE is changed.

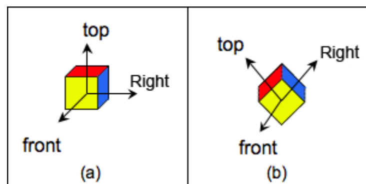

Fig. 18. (a) default external orientation; (b) external orientation 45 degrees counterclockwise around front axis

Example. The following F-logic statement defines the concept WeldingRobot:

WeldingRobot :: concept

The concept *WeldingRobot* may have one or more properties. Suppose it has a weight property with default value 1500 and a color property with default value 'red'. All properties of a concept should 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. Let's go back to our example. The properties for WeldingRobot should be defined as follows:

WeldingRobot[weight $\bullet\!\!\rightarrow$ *1500;*
color $\bullet\!\!\rightarrow$ *red]*

An instance of a domain concept will be represented in the F-logic as an instance of the corresponding F-logic class.

Example. The following F-logic statement defines an instance *WeldingRobot1* of the domain concept *WeldingRobot*:

WeldingRobot1 : WeldingRobot

To overwrite the default value for color, we use the following statement: *WeldingRobot1[color → green]*

Complex Concept. Remember that a complex concept consists of a number of components. Components can be simple concepts or complex concepts. To specify that a concept is a component of some other concept, a $partOf$ property is used.

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:
$a[partOf \Rightarrow b]$

Each complex concept also has a $reference$ *component*. The reference component is used for the positioning and orientation of the complex concept. The position and orientation of the reference component will also be the position and orientation of the complex concept. In addition, all other components of the complex concept will be positioned and oriented relative to this reference component (according to the specifications given by the connection relations). This does not imply that the components cannot be moved anymore. If it is not forbidden by constraints and/or connections, the components can still be moved. In this case, the relative position and the orientation of the components will change.

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 component of a complex concept, then this concept should also be a component of this complex concept.
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 are used to define the modeling concept $complexConcept$. It is defined as a subclass of the *concept* class (since a complex concept is also a concept) with properties (methods) $allParts$ and $referencePart$

$complexConcept :: concept$
$complexConcept[allParts \Rightarrow\!\!\!> concept;$
$\quad\quad referencePart \Rightarrow concept]$

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

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

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

The position (method) for a complex object is defined as follows:

$$\begin{aligned} A[position \rightarrow P] \leftarrow\ & A{:}complexConcept \wedge C{:}concept \wedge P{:}point \wedge \\ & C[referencePartOf \rightarrow A] \wedge \\ & C[position \rightarrow P] \end{aligned}$$

Example. The following F-logic statements define a WeldingRobot as a complex concept composed of a Rail, a Base, a LowerArm, an UpperArm, and a WeldingHead. The Rail is used as reference component.

$WeldingRobot :: complexConcept$
$Rail :: concept$
$Base :: concept$
$LowerArm :: concept$
$UpperArm :: concept$
$WeldingHead :: concept$
$Rail[partOf \Rightarrow WeldingRobot]$
$Base[partOf \Rightarrow WeldingRobot]$
$LowerArm[partOf \Rightarrow WeldingRobot]$
$UpperArm[partOf \Rightarrow WeldingRobot]$
$WeldingHead[partOf \Rightarrow WeldingRobot]$
$Rail[referencePartOf \Rightarrow WeldingRobot]$

To create an instance of the complex concept, the following F-logic statements could be used:

$WeldingRobot1 : WeldingRobot$
$Rail1 : Rail$
$Base1 : Base$
$LowerArm1 : LowerArm$
$UpperArm1 : UpperArm$
$WeldingHead1 : WeldingHead$
$Rail1[partOf \rightarrow WeldingRobot1]$
$Base1[partOf \rightarrow WeldingRobot1]$
$LowerArm1[partOf \rightarrow WeldingRobot1]$

UpperArm1[partOf $\rightarrow$ *WeldingRobot1]*
WeldingHead1[partOf $\rightarrow$ *WeldingRobot1]*
Rail1[referencePartOf $\rightarrow$ *WeldingRobot1]*

If we would now like to know the position of the instance $WeldingRobot1$ in the VE, we could use the position method from the *concept* class (using tools like OntoBroker [35] or Flora-2 [36]).

In a similar way, we can define the external and internal orientation of a complex concept. Details are omitted.

So far we have shown how complex concepts are formalized. Note that all methods are working on instances of complex concepts. However, it may also be useful to be able to query the concept level and ask the system about the concepts that are part of some complex concept. Therefore we have overloaded the methods $allParts$ and $referencePart$ so that they also work on classes:

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

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

Concepts themselves don't appear in the VE (only the instances), therefore they don't have a position and neither an orientation. Therefore, there is no need to overload the methods *position*, *externalOrientation* and $InternalOrientation$.

4.4 Formalization of the Connection Relations

In this section we will explain how the connection relations are formalized. Note that the semantics of the connection relations can be considered from two different viewpoints. We can consider the *axiomatic semantics*, which only defines the specific properties (expressed as assertions) of the effect of using the connection relation. However, it is also possible to consider the *operational semantics*, which defines the meaning of the connection relation by the computation it induces when it is used. Expressing the axiomatic semantics is easier as the aspects of the execution are ignored. For example, for a connection axis relation it would be sufficient to state that the two connection axes need to coincide. For the operational semantics, it is also necessary to specify how this should be realized. This makes it much more complex, however, such a semantics has the advantage that the formal specifications can also be used to calculate actual positions and orientations and to reason about this when objects are actual moved or manipulate inside the VE. As we have plans to use our formalization for this purpose, we decided to use the operational semantics.

Using the operational semantics, we also had to distinguish between what we call the *initial semantics* and the *simulation semantics*. On the one hand, a

connection relation expresses how two components should be actually 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 relation at the time it is used to connect two components. On the other hand, a connection relation also expresses constraints on the possible motions of the components during the rest of the lifetime of the connection. This is what we will call the simulation semantics.

Because of the complexity introduced by the operational semantics, we will only give the formalization for the connection axis relation and also not completely elaborate all details. Details can be found in [5].

Formalization of the connection relations. Remember that the connection axes are defined using 3 predefined planes: the horizontal plane, the vertical plane and the perpendicular plane. These planes are defined as subclasses of plane, a class without properties. The definition of the horizontal plane is given below. As we have seen earlier, the default horizontal plane can be rotated over the left-to-right axis and over the front-to-back axis. This is expressed by the properties $leftToRightAngle$ and $frontToBackAngle$ (with default value 0). The position is given by the properties x0, y0 and z0 (default value 0). Translating the horizontal plane along the top-to-bottom axis is expressed by overwriting the z0 property. The vertical plane and the perpendicular plane are defined in a similar way.

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

Now we are ready to formalize the connection axis relation. First we will formalize the initial semantics of the connection axis relation. The *connectionAxis Relation* class contains a number of properties to identify for each component the two planes used to define the connection axis ($sourcePlane1$, $sourcePlane2$, $targetPlane1$, and $targetPlane2$). There are also a number of properties that specify the connection point for the source and the target component ($sourceTP Dist$, $sourceTPDir$, $targetTPDist$, $tragetTPDir$). Remember that a connection point is specified by a distance and a direction. The $connectionAxisRelation$ class also has two properties $sourceAxis$ and $targetAxis$ which are methods that return the actual axis (a line) which is the connection axis for the source, respectively the target component.

connectionAxisRelation [*sourcePlane1* $\Rightarrow$ *plane;*
sourcePlane2 $\Rightarrow$ *plane;*
targetPlane1 $\Rightarrow$ *plane;*
targetPlane2 $\Rightarrow$ *plane;*

sourceAxis $\Rightarrow$ *line;*
targetAxis $\Rightarrow$ *line;*
sourceTPDist $\Rightarrow$ *float;*
sourceTPDir $\Rightarrow$ *string;*
targetTPDist $\Rightarrow$ *float;*
targetTPDir $\Rightarrow$ *string]*

The fact that two components are connected by means of a connection axis relation is given by means of the *connectTo* property:

Let a and b be two concepts (a:: concept and b:: concept). The fact that a is connected to b by means of a connection axis relation r (r : *connectionAxis Relation*) *is expressed as follows:*

a [*connectedTo@b* $\bullet\!\!\rightarrow$ r]

The method *sourceAxis* is defined as follows:

C[*sourceAxis* $\rightarrow L$] $\leftarrow$ C : *connectionAxisRelation* $\wedge$ L : *line* $\wedge$
A[*connectedTo@B* $\bullet\!\!\rightarrow C$] $\wedge$
A[*position* $\rightarrow p$] $\wedge$
intersectionLine(C.sourcePlane1,
C.sourcePlane2, M) $\wedge$
L : *line*[$x_0 \rightarrow M.x_0 + p.x;$
$y_0 \rightarrow M.y_0 + p.y;$
$z_0 \rightarrow M.z_0 + p.z;$
$a \rightarrow M.a;\ b \rightarrow M.b;\ c \rightarrow M.c$]

The predicate *intersectionLine* is used to state that a line M is the intersection of two planes. The definition of this predicate is omitted. p is the position of the source component. L (the connection axis) express a line through the point p and parallel with M (given by parametric equations).

The method *targetAxis* is defined in a similar way as *sourceAxis* (source should be replaced by target).

Now, we can define the initial semantics of a connection axis relation in terms of the position and orientation of the source component. This is done by means of the predicates *connectedPosition*(A, C, P) and *connectionAxisPos*(A, C, P).

The predicate *connectedPosition*(A, C, P) states that P is the position of a concept A connected via a connection relation to a concept C. The predicate works for A being the source of a connection relation as well as for A being the target. Note that the predicate can be defined in such a way that it applies for whatever type of connection relation used to make the connection between A and C. However, to keep it simple, the definition given here only takes a connection axis relation into account.

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))

The predicate $connectionAxisPos(A, C, P)$ states that P is the position of A when it is connected via a connection axis relation to C, while the predicate $TPPos$ states that P is the position of A when it is playing the role of target object in the connection axis relation R. Details of this predicate are omitted.

The predicate $connectedOrientation(A, C, E)$ is defined to state that E is the external orientation for A connected via a connection relation to a concept C.

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)*

The predicate $connectionAxisOrient(A, C, E)$ states that E is the external orientation of A when it is connected via a connection axis relation to C. Details of this predicate are omitted.

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 target's 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 relations still coincide.

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*]*

The predicate $caPosConstraint(A, C, P)$ states that P is the position of A when it is connected via a connection axis relation to C. Note that this predicate is very similar to the predicate *connectionAxisPos*. However, for the predicate *connectionAxisPos* the source is positioned along the connection axis according to the specified translation point. Now for the simulation semantics, the translation points are not relevant anymore. Therefore a different predicate was needed. We omit the details of $caPosConstraint$.

5 Related Work

This section will discuss related work. The first subsection reviews a number of academic modeling approaches. The second subsection reviews a number of languages that supports the modeling of rigid-body and the third subsection reviews a number of commercial modeling approaches.

5.1 Academic Modeling Approaches

Kim et al. propose a structural approach for developing VR applications [40]. The approach is called ASADAL/PROTO and it uses 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. However, there is no support in VOS to describe physical connections and constraints between different objects.

The CODY Virtual Constructor [41] [42] [43] is a system which enables an interactive simulation of assembly processes with construction kits in a virtual environment. The assembly process happens either by means of direct manipulation or by means of natural language. 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. For the representation of the knowledge inside the knowledge bases a framebased representation language COAR (Concepts for Objects, Assemblies and Roles) has been developed. 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. Another disadvantage is that the connection ports must be defined in advance. 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.

The aim of the Open Assembly Model (OAM) [44] 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 [45]. 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. An Assembly is a composition of its subassemblies or parts. OAM has been designed to represent information used or generated inside CAD3-like tools. 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.

The Virtual Assembly Design Environment (VADE) [46] [47] is a VR-based engineering application that allows engineers to evaluate, analyze, and plan the

assembly of mechanical systems. 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. The VADE system is intended for engineers and it is far from high-level.

The Multi-user Intuitive Virtual Environment (MIVE) [48] [49] [50] provides a simple way for objects to constrain to each other without having to use a complete constraint solver. MIVE uses the concept of 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. 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.

5.2 Languages Supporting Rigid-Body

Recently, a revision of the X3D architecture [51] 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 and allow one body's motion to effect another. Examples of joints offered are BallJoint, SingleAxisHingeJoint, SliderJoint or UniversalJoint. Although X3D is sometimes entitled as being high-level, it is still focussed only on what to render in a scene instead of how to render the scene. X3D is still not intuitive for a non-VR expert as for example, he still needs to specify a hinge constraint by means of points and vectors. Furthermore, reasing over X3D specifications is far from easy.

Collada [52] also has a rigid body specification in the same way as X3D. But Collada has been created as an independent format to describe the 3D content that can be read by any software. Therefore, Collada is a format for machine and not really for human and certainly not for describing the modeling of complex objects from a high-level point of view.

5.3 Commercial Modeling Approaches

SimMechanics [53] is a set of block libraries and special simulation features to be used in the Simulink environment. Simulink is a platform for simulation in different domains and model-based design for dynamic systems. It provides an interactive graphical environment that can be used for building models. It contains 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. Although SimMechanics

is already on a higher-level of abstraction (than for example physics engine programming), it is still too low-level to be generally usable for application domain experts. Another disadvantage of the approach is that there is no possibility to do some reasoning.

SolidWorks [54] 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. MotionWorks contains different families of joints. It is fully integrated into Solid-Works. These tools are not suited for non-experts. The vocabulary used in SolidWorks is really meant for the expert.

3D Studio Max (3ds max) [20] is a 3D modeling software in the category of authoring tools. There are also other tools ([24], Maya[21]) similar to 3ds max and therefore this paper will review 3ds max as a representative of this category. 3ds max provides grouping features which enable the user to organize all the objects with which he is dealing. 3ds max has also another way of organizing objects, namely by building a linked hierarchy. 3ds max provides a number of constraints that can be used to force objects to stay attached to another object. Although 3ds max is intended to create virtual environments without the need for detailed programming, 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 layman.

6 Conclusions and Further Work

In this paper, we have described why conceptual modeling can be important for the field of VR. We also explained the shortcomings of current conceptual modeling languages with respect to VR. Next, we have presented a conceptual modeling approach for VR and we have introduced conceptual modeling concepts to specify complex connected 3D objects. The semantics of the modeling concepts presented are defined formally using F-logic, a full-fledged logic following the object-oriented paradigm. Operational semantics have been defined for the modeling concepts. The use of operational semantics has the advantage of being able to actually calculate positions and orientations of objects and to reason about the specifications. The use of F-logic also allows using a single language for specifying the three different levels involved in our approach, i.e. the meta-level, the concept-level and the instance-level, as well as for querying.

We do not claim that the conceptual modeling concepts presented here are a complete set of modeling primitives for 3D complex objects. One limitation concerning the modeling of connections is that it is not yet possible to define a connection between two components that is a combination of connections, or to combine constraints on connections. This is needed to allow for more powerful

connections such as e.g., a car wheel. For this we need actually a combination of two hinge constraints. The problem however is that the motion allowed by one hinge constraint may be in contradiction with the motion allowed by the second hinge constraint, and therefore a simple combination is not sufficient. Next, we also need modeling concepts for specifying so-called contact joints. This type of joints does not really describe a connection but rather a contact between two objects like the gearwheels of a watch that need to roll against each other.

Another limitation is that the approach presented here is only usable for modeling VE's up to a certain level of complexity. E.g., the approach 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 different domains. However, our approach can always be used for fast prototyping. Prototype tools [55] are developed that allow generating code from the (graphical) conceptual specifications. Afterwards, VR experts using other tools such as VR toolkits may refine the prototype generated from the conceptual specifications.

The formalization given, unambiguously defines the modeling concepts. However, in order to use it for reasoning and consistency checking, an implementation needs to be built. I.e. the conceptual specifications, which are given by a designer using the graphical notation, need to be translated into their corresponding F-logic representation and added to some knowledge bases. Having all the information in F-Logic knowledge bases, we can then use existing F-logic systems (possibly extended with extra features), such as Flora-2, to query the conceptual specifications and to do some reasoning and consistency checking. Currently, we are working on such a reasoning system. It allows specifying a number of domain-independent as well as domain-dependent rules. Examples of domain-independent rules are: a rule to ensure that the partOf relation is anti-symmetric, a rule to detect complex objects that don't has a reference object, or a rule that detects objects that are placed at the same location. Examples of domain-specific rules are: a rule that specifies that all robot-instances should be positioned on the ground, or a rule that each robot should have a controller. More on this can be found in [56].

For future work we plan to extend the implementation to be able to dynamically update the conceptual specifications at run-time. Therefore we need to implement a mechanism so that changes in the actual VE are directly reflected in the logical representation. Such an extension would also allow querying VE's in real time.

Acknowledgment

The work presented in this paper has been funded by FWO (Research Foundation - Flanders) and partially also by IWT (Innovation by Science and Technology Flanders).

References

1. Second Life, http://www.secondlife.com/
2. Fowler, M., Kendall, S.: UML Distilled: a brief introduction to the standard object modeling language. Addison-Wesley Professional, London (1999)
3. Halpin, T.: Conceptual Schema and Relational Database Design. WytLytPub (1999)
4. Halpin, T.: Information Modeling and Relational Databases: From Conceptual Analysis to Logical Design. Morgan Kaufmann, San Francisco (2001)
5. Bille, W.: Conceptual Modeling of Complex Objects for Virtual Environments, A Formal Approach. Ph.D. thesis, Vrije Universiteit Brussel, Brussels, Belgium (2007)
6. Bille, W., De Troyer, O., Kleinermann, F., Pellens, B., Romero, R.: Using Ontologies to Build Virtual Worlds for the Web. In: Proc. of the IADIS International Conference WWW/Internet 2004 (ICWI 2004), Madrid, Spain, pp. 683–689. IADIS Press (2004)
7. De Troyer, O., Bille, W., Romero, R., Stuer, P.: On Generating Virtual Worlds from Domain Ontologies. In: Proc. of the 9th International Conference on Multi-Media Modeling, pp. 279–294 (2003)
8. De Troyer, O., Kleinermann, F., Pellens, B., Bille, W.: Conceptual Modeling for Virtual Reality. In: Tutorials, Posters, Panels, and Industrial Contributions of the 26th international Conference on Conceptual Modeling (ER 2007), CRPIT, Auckland, New Zealand, pp. 5–20 (2007)
9. Kleinermann, F., De Troyer, O., Mansouri, H., Romero, R., Pellens, B., Bille, W.: Designing Semantic Virtual Reality Applications. In: Proc. of the 2nd INTUITION International Workshop, pp. 5–10 (2005)
10. Vince, J.: Introduction to Virtual Reality. Springer, Heidelberg (2004)
11. Burdea, G., Coiffet, P.: Virtual Reality Technology. John Wiley & Sons, Chichester (2003)
12. Activeworlds, http://www.activewords.com
13. Quake, http://www.idsoftware.com/gmaes/quake/quake
14. Unreal, http://www.unreal.com/
15. Hartman, J., Wernecke, J.: The VRML 2.0 Handbook. Addison Wesley, London (1998)
16. Web 3D Consortium, http://www.web3d.org/x3d/specifications/
17. Selman, D.: Java3D Programming. Manning (2002)
18. Octaga, http://www.octaga.com
19. Vivaty, http://www.vivaty.com/
20. Kelly, L.: Murdock, 3ds max 5 bible. Wiley Publishing, Chichester (2003)
21. Derakhshami, D.: Introducing Maya 2008. AutoDesk Maya Press (2008)
22. Milkshape, http://chumbalum.swissquake.ch
23. AC3D, http://www.invis.com/
24. Roosendaal, T., Selleri, S.: The official Blender 2.3 Guide: Free 3D creation suite for Modeling, Animation and rendering. No Starch Press (2005)
25. Performer, http://www.sgi.com/products/software/performer
26. X3D Toolkit, http://artis.imag.fr/Software/x3D/
27. Xj3D, http://www.xj3d.org
28. Chen, P.: The Entity-Relationship Model: Towards a Unified View of Data. ACM Transactions on Database Systems 1(1), 471–522 (1981)

29. Pellens, B.: A Conceptual Modelling Approach for Behaviour in Virtual Environments using a Graphical Notation and Generative Design Patterns. Ph.D. thesis, Vrije Universiteit Brussel, Brussels, Belgium (2007)
30. Pellens, B., De Troyer, O., Bille, W., Kleinermann, F.: Conceptual modeling of object behavior in a virtual environment. In: Proceedings of Virtual Concept, pp. 93–94 (2005)
31. Pellens, B., De Troyer, O., Bille, W., Kleinermann, F., Romero, R.: An ontology-driven approach for modeling behavior in Virtual Environments. In: Meersman, R., et al. (eds.) Proceedings of Ontology Mining and Engineering and its use for Virtual Reality, pp. 1215–1224. Springer, Heidelberg (2005)
32. Pellens, B., Kleinermann, F., De Troyer, O., Bille, W.: Model-based design of virtual environment behavior. In: Zha, H., et al. (eds.) Proceedings of the 12th International Conference on Virtual Reality Systems and Multimedia, pp. 29–39. Springer, Heidelberg (2006)
33. Pellens, B., De Troyer, O., Kleinermann, F., Bille, W.: Conceptual modeling of behavior in a virtual environment. International Journal of Product and Development, Inderscience Enterprises, 14–18 (2006)
34. Pellens, B., Kleinermann, F., De Troyer, O.: Intuitively Specifying Object Dynamics in Virtual Environments using VR-WISE. In: Proc. of the ACM Symposium on Virtual Reality Software and Technology, pp. 334–337. ACM Press, New York (2006)
35. Ontoprise GmbH.: How to write F-Logic Programs covering OntoBroker version 4.3 (2006)
36. May, W.: How to Write F-Logic Programs in Florida. Institut fur Informatik. Universitat Freiburg, Germany (2006), http://dbis.informatik.uni-freiburg.de
37. ODE, http://www.ode.org/
38. PhysX, http://www.nvidia.com/object/nvidia_physx.html
39. MotionWork, http://www.motionworks.com.au/
40. Kim, G.J., Kang, K.C., Kim, H.: Software engineering of virtual worlds. In: Proceedings of the ACM symposium on Virtual Reality and Technology, pp. 131–138. ACM Press, New York (1998)
41. Wachsmuth, I., Jung, B.: Dynamic conceptualization in a mechanical-object assembly environment. Artificial Intelligence Review 10(3-4), 345–368 (1996)
42. Jung, B., Hoffhenke, M., Wachsmuth, I.: Virtual assembly with construction kits. In: Proceedings of ASME Design Engineering Technical Conferences, pp. 150–160 (1997)
43. Jung, B., Wachsmuth, I.: Integration of geometric and conceptual reasoning for interacting with virtual environments. In: Proceedings of the AAAI Spring Symposium on Multimodal Reasoning, pp. 22–27 (1998)
44. Rachuri, S., Han, Y.-H., Foufou, S., Feng, S.C., Roy, J., Wang, F., Sriram, R.D., Lyons, K.W.: A model for capturing product assembly information. Journal of Computing and Information Science in Engineering 6(1), 11–21 (2006)
45. Fenves, S.: A core product model for representing design information. Technical report NISTIR 6736, National Institute of Standards and Technology, NIST (2001)
46. Jayaram, S., Connacher, H., Lyons, K.: Virtual assembly using virtual reality techniques. Journal of Computer-Aided Design 29(8), 155–175 (1997)
47. Jayaram, S., Wang, Y., Jayaram, U., Lyons, K., Hart, P.: A virtual assembly design environment. In: Proceedings of IEEE Virtual Reality Conference, pp. 172–180 (1999)
48. Smith, G., Stuerzlinger, W.: Integration of constraints into a vr environment. In: Proceedings of the Virtual Reality International Conference, pp. 103–110 (2001)

49. Gosele, M., Stuerzlinger, W.: Semantic constraint for scene manipulation. In: Proceedings of the Spring Conference in Computer Graphics, pp. 140–146 (2002)
50. Stuerzlinger, W., Graham, S.: Efficient manipulation of object groups in virtual environments. In: Proceeding of the VR 2002 (2002)
51. Brutzman, D., Daly, L.: X3D: Extensible 3D graphics for Web Authors. Morgan Kaufmann, San Francisco (2007)
52. Arnaud, R., Brnes, M.: Collada: Sailing the gulf of 3d digital content creation. A K Peters, Ltd., Massachusetts (2006)
53. SimMechanics, `http://www.mathworks.com`
54. SolidWorks, `http://www.solidworks.com`
55. Coninx, K., De Troyer, O., Raymaekers, C., Kleinermann, F.: VR-DeMo: a Tool-supported Approach Facilitating Flexible Development of Virtual Environments using Conceptual Modelling. In: Proc. of Virtual Concept, pp. 65–80 (2006)
56. Mansouri, H., Kleinermann, F., De Troyer, O.: Detecting Inconsistencies in the Design of Virtual Environments over the Web using Domain Specific Rules. In: Proc. of the 14th International Symposium on 3D Web Technology (Web3D 2009), pp. 31–38. ACM Press, New York (2009)