When Model Driven Engineering meets Virtual Reality: Feedback from Application to the Collaviz Framework

Thierry Duval, Arnaud Blouin, Jean-Marc Jézéquel

▶ To cite this version:

Thierry Duval, Arnaud Blouin, Jean-Marc Jézéquel. When Model Driven Engineering meets Virtual Reality: Feedback from Application to the Collaviz Framework. Software Engineering and Architectures for Realtime Interactive Systems Working Group, Mar 2014, Minnesota, United States. ⟨hal-00969072⟩

HAL Id: hal-00969072

https://inria.hal.science/hal-00969072v1

Submitted on 2 Apr 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

HAL Authorization

To appear in SEARIS 2014

When Model Driven Engineering meets Virtual Reality: Feedback from Application to the Collaviz Framework

Thierry Duval*
Université Rennes 1
IRISA – UMR CNRS 6074

Arnaud Blouin†
INSA Rennes
IRISA – UMR CNRS 6074

Jean-Marc Jézéquel‡
Université Rennes 1
IRISA – UMR CNRS 6074

Abstract

Despite the increasing use of 3D Collaborative Virtual Environments (3D CVE), their development is still a cumbersome task. The various concerns to consider (distributed system, 3D graphics, *etc.*) complexify the development as well as the evolution of CVEs. Software engineering recently proposed methods and tools to ease the development process of complex software systems. Among them, Model-Driven Engineering (MDE) considers models as first-class entities. A model is an abstraction of a specific aspect of the system under study for a specific purpose. MDE thus breaks down a complex system into as many models for different purposes, such as: generating code from models; building domain specific programming/modeling languages (DSL); generating tools such as graphical or textual editors. In this paper we leverage MDE for developing 3D CVEs. We show how the Collaviz framework took benefits from a DSL we built. The benefits are multiple: 3D CVE designers can focus on the behavior of their virtual objects without bothering with distributed and graphics features; configuring the content of 3D CVEs and their deployment on various software and hardware platforms can be automated through code generation. We detail the development process we propose and the experiments we conducted on Collaviz .

Keywords: Virtual Reality, Collaborative Virtual Environments, Frameworks, Software Engineering, Model Driven Engineering

Index Terms: H.5.2 [Information Interfaces and Presentation (e.g., HCI)]: User Interfaces—Theory and methods; H.5.3 [Information Interfaces and Presentation (e.g., HCI)]: Group and Organization Interfaces—Computer-supported cooperative work (CSCW); I.3.7Computer Graphics3-Dimensional Graphics and RealismVirtual reality; D.2.11 [Software Engineering]: Software Architectures—Patterns; I.3.6 [Computer Graphics]: Methodology and Techniques—Interaction techniques, Device independence

1 Introduction

3D Collaborative Virtual Environments (CVE) are now widely used since the broadening of high bandwidth networks and 3D graphics on any kind of devices. Examples of devices are the CAVE immersive systems [5], powerful workstations, laptops, or even mobile devices such as tablets or smartphones. Yet 3D CVEs are still difficult to develop since they mix difficulties coming from two different areas: 3D graphics and networking, as well as multiple roles, from system programming to virtual object and virtual environment (VE, the game play) designers.

We can identify 3 current issues hindering the development of 3D CVEs. First, the design of 3D CVEs merges the design of interactive 3D applications and distributed collaborative applications. This task is complex since it must address 3D interaction and immersion issues as well as collaborative ones dealing with distribution, synchronization, and consistency maintenance of the shared VE. This is why CVE frameworks should provide users with facilities allowing them to focus on the behavior of the shared virtual objects of CVEs rather than having to manage 3D graphics or distribution and synchronization issues.

Second, CVE frameworks should also allow CVE designers to describe the content of their shared universes, making the shared virtual objects they designed easier to instantiate and configure.

Last, the configuration (adaptation to the hardware deployment systems) of CVEs is complex since it must address various network characteristics (from high bandwidth on professional or experimental networks to low bandwidth on personal networks) as well as various displays and 3D interaction devices (from 6-face *Cave-like systems* to simple workstations or even to simple interactive tablets). All these configurations can even be used at the same time in a single deployment. For instance, that would permit to perform asymmetric collaboration between remote users using different input and output devices. So, CVE frameworks should also provide users with facilities allowing them to easily configure how a CVE would be deployed on the rendering devices that would be available at run-time.

To tackle these 3 current limitations, CVE frameworks should provide different tools at different conception stages to help the virtual object designers, the VE designers, and the staff in charge of the deployment of VEs on specific hardwares. These tools are: dedicated editors for editing virtual objects and environments; code generators for alleviating cumbersome code development tasks, thus helping designers to focus on their core concerns.

Model-Driven Engineering (MDE) [19] is a paradigm that considers models as first-class entities. Models represent reality for the given purpose. MDE permits to deal with systems in a simplified manner by breaking down a complex system into as many models as needed. Then, models can be used for various purposes such as code generation, verification, interoperability between tools. In this paper we show how MDE can be leveraged for the development of CVEs by proposing an MDE-based development toolchain. We validate our proposal with an experiment we conducted on the Collaviz framework.

The paper is organized as follows. Section 2 presents the context of our work: the need to design CVEs with various distribution models and 3D graphics API at different levels (virtual objects, shared environment, adaptation to hardware). Section 3 presents the Open Source Collaviz CVE framework, how it deals with distribution modes and 3D graphics API, and how it can be used at these 3 levels. Section 4 introduces a motivating example illustrating our problematic. Section 5 gives some background on Model-Driven Engineering (MDE). Section 6 presents the interest of MDE for the design of CVEs. Section 7 details an experiment we conducted consisting of applying an MDE process within the toolchain of the Collaviz framework.

*e-mail: thierry.duval@irisa.fr
†e-mail: arnaud.blouin@irisa.fr
‡e-mail: jean-marc.jezequel@irisa.fr

DRAFT

2 Related Work

In [20] Steed shows that *within the virtual environments community there is a large cost of maintaining software demonstrations and applications whilst hardware and low-level software changes*. He gives the example of his own lab, where, over a period of 15 years, people developed software using at least 40 VE software systems. Then, he gives important cues about how to write Virtual Reality (VR) systems that could remain operational over the years, such as: loosely-coupling virtual objects together (preferring sending events and messages rather than invoking methods); decoupling interface from application; making high level abstractions for the users, the navigation model, or the network. The DIVE framework [10] follows these requirements and already demonstrated its capability to develop VR software systems for more than 10 years. Similar to DIVE, the VR-DECK Toolkit [2] is a development environment consisting of C++ class libraries for module construction, interprocess communication, device support, and hierarchical object-oriented graphics. It allows its users to create virtual environments using a mixed object-oriented and event-based paradigm for defining system behavior. Modules represent entities in the world such as objects, operations, functions, or users. Modules communicate with each other by producing and consuming events. They are also defined at a high-level using rules written in C++ that determine how events are handled. The run-time environment includes: a 2D user interface for dynamically constructing worlds from a collection of modules, allocating processes among hosts, editing modules, and monitoring operation; a library of ready-to-use modules for various devices and common operations. Making such high-level abstraction and separating concerns between graphics and behaviors (such as recommended by the MVC [17][11] and the PAC [4] models) have demonstrated their efficiency for 3D interaction design [3]. However, this kind of approach is still too strongly coupled to a programming language.

The Virtual Reality Modeling Language (VRML) [12] and its successor X3D[1] aim at providing a high-level description of a 3D VEs. They do not make any assumption about the hardware devices that would be used at run-time neither for the rendering of the VE nor for the interactions. These approaches mainly focus on the geometrical organization of a VE, its animation through *interpolators*, and provide very little interaction facilities with dedicated *sensors*. Providing advanced behaviors for virtual objects and for interactions must be done using Javascript or Java additional code linked with VRML nodes, which can make its evolution difficult, especially for using dedicated input devices. Some authoring tools such as *BS Content Studio*[2] enable VE designers to quickly develop VE described with VRML or X3D. These tools also offer facilities to edit the features of VRML or X3D nodes and events, including the expression of *ROUTEs* between VRML outputs and inputs for animation and interaction. Nevertheless, this approach mostly concerns Web3D application deployment, without considering neither collaborative features nor adaptation to specific hardware devices.

MiddleVR[3] is a generic immersive VR framework. It does not focus on the creation of VR contents or 3D interactions but on the configuration of VR systems. It proposes a graphical configuration tool to setup a VR system by handling hardware input and output devices. It relies on VRPN [21] for the integration of some input devices, or on direct integration of other device SDKs. This interesting approach is complementary to usual tools dedicated to the design of VEs.

Contigra (Component OrieNted Three-dimensional Interactive GRaphical Applications) [6] is a declarative XML approach that allows to describe 3D components implementation (defining new component classes), configuration (instantiating components), and composition (assembling and linking components) through 3D editors (component builders and scene builder tools). The Interaction Techniques Markup Language (IntML) [8] is designed for non-programmers who plug VR objects, devices, and 3D interaction techniques together, by binding outputs to inputs, to design a new application only by XML coding. Then IntML files can be compiled toward core frameworks or API such as Java3D or VR-Juggler if the IntML binding exists with them. Both approaches focus mainly on the creation of XML files, and do not cover neither the design of virtual objects complex 3D behavior nor distribution and networking features.

The *Scene Structure and Integration Modeling Language* (SSIML) [22] proposes a model-driven approach that uses the SSIML model as a contract between three kinds of developers: the software designer, the 3D content developer, and the programmer of a VR application. The software designer creates the SSIML model and uses two code generators. The first one produces the 3D scene code, that a 3D content developer has to complete with a 3D authoring tool. The second one produces program and behavior code, that a programmer has to complete with a programming tool. The same approach has been adapted to the development of Web3D applications using SSIML, X3D and Javascript [15]. This interesting distribution of tasks among different actors with different skills can be adapted to our own design process but must be extended to take into account hardware and network features of CVEs.

The MASCARET framework [1] aims at developing Intelligent VEs. This framework provides an extension of the *Unified Modeling Language* (UML)[4] to support several CVE concerns. Thanks to this extension, domain experts can define structural (class diagram) and behavioral (e.g. state machine) concerns of a VE. Then, 3D designers can import these definitions into a 3D modeler thanks to a dedicated plug-in to construct objects. While this approach is conceptually close to ours, several differences remain. First, the core idea of our proposal is to help CVE developers in their coding tasks. We thus generate editors and code in this purpose. On the contrary, MASCARET extends existing languages and tools for the design of semantic VR environments. Second, one feature of our approach is the generation of editors dedicated to a specific VE, while MASCARET extends existing tools. This difference implies pros and cons for each approach. For instance editors generated using our approach are more tailored to the domain at hands than MASCARET editors. However, these generated editors may have some specific features missing.

3 The Collaviz framework

For this study we chose the open-source Collaviz framework[5] introduced in this section.

3.1 Roles

The Collaviz framework involves 4 different complementary roles:

1. concept creators design virtual objects
2. world instantiators (scene designers) use these virtual objects to create virtual worlds
3. physical world connectors map virtual world with physical devices to allow users to interact with the designed CVE
4. users interact with virtual worlds for various purposes (*e.g.* playing games, architecting an apartment)

In this work we focus on roles that involve some development tasks, *i.e.* the 3 first ones as detailed in the following sub-sections.

[1] `http://www.web3d.org/x3d/`
[2] `http://www.bitmanagement.com/`
[3] `http://www.imin-vr.com/middlevr/`
[4] `http://www.uml.org/`
[5] Collaviz is publicly available at `www.collaviz.org`

DRAFT

3.2 Designing new virtual objects

When designing shared virtual objects, the Collaviz framework uses a PAC-C3D [7] architecture to ensure an efficient separation between the behavior of these objects and their network (distribution and synchronization) and 3D graphics features.

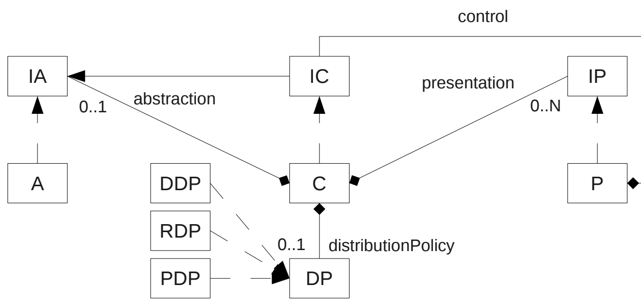

Figure 1: The PAC-C3D architectural model for CVE

When designing new virtual objects according to this software architectural model, a designer must create at least 7 new Java classes and interfaces: *IA*, *A*, *IP*, *P*, *A*, *ClientC*, and *ServerC*. These two different kinds of controllers are needed because of the Collaviz network architecture [9] that uses a central public server to ensure that every client can connect to a shared VE. As only the machine that hosts this server has to be known by the other machines, Collaviz uses different controllers for virtual object according to their network location, independently of their role (*referent* or *proxy*), and their distribution policy that are provided and managed by the framework.

Nevertheless, this allows a designer to focus on the behavior of virtual objects through the *A* component. It also makes him create a lot of other components that he is not really interested in. Moreover, even this behavior can be difficult to code because of the patterns that the designer must use according to declare and access the parameters of these virtual objects.

3.3 Designing new shared virtual environments

Once virtual objects available, the Collaviz framework allows VE designers to describe the content of their VEs through instantiations of these virtual object classes. This description is achieved in several XML files, which contain all the details (parameter values) needed to instantiate properly these virtual objects. VE designers can be someone other than virtual objects designers, their role is similar to the role of a gameplay designer for a 3D video game.

So, the description of the content of a VE is possible without any java programming. Nevertheless, VE designers must know the types and ranges of values that must be provided for a proper instantiation of each type of virtual object.

3.4 Deploying collaborative applications

The last description step that must be achieved to obtain a CVE is the hardware setup. This step consists in associating physical input devices to logical interaction tools. Indeed, the Collaviz framework makes it possible to describe 3D interactions with a high level of abstraction. It is possible by programming these interactions through the use of high-level interaction events described in the abstraction layer. That can be done independently from the physical input devices that will be used at run-time to drive these logical interaction tools, assuming that drivers exists to dialog with these hardware input devices. Here again, these associations are described in several XML files. Each participant who will join a CVE will need such files dedicated to adapt his interaction possibilities to his available input devices. These descriptions files can be filled by yet another kind of designer in charge of the adaptation of the interactions to the available hardware. Once again, these designers do not need to program, but they must know some precise details about the drivers of the hardware input devices and about how to associate their outputs to some logical interaction events.

3.5 The next step

So, designing a CVE framework following an architecture as PAC-C3D is only the first step to allow a developer to focus only on the application aspects of his CVE. A CVE framework should also provide tools and processes to limit the need of coding 3D graphics (the *P* components) or network aspects of virtual objects (the *ClientC* and *ServerC* components). Similarly, VE designers should be guided while describing virtual objects in XML files to avoid some errors when setting parameters or values. Last, similar guiding should be provided when other designers describe how end-users' hardware devices must be associated to higher-level interaction descriptors such as logical input events.

4 Motivated Example

In this section, we detail a VE example to motivate the current limitations in the development process of CVEs. We will consider the design of a kind of *compass* that can aim at a target that it should be possible to change. Nearly all the code samples that are presented in the section will be automatically generated with the tools obtained through the MDE approach detailed in section 6. Only the **compute** method presented at the end of subsection 4.1.1, and the **update** method presented at the end of subsection 4.1.3 must be coded manually. **These code samples appear boldface to make it easy to distinguish them from the code that can be generated through the MDE approach**.

4.1 Designing the virtual object

The first step of the development process consists of designing the virtual objects. A compass virtual object can be inherited from the Collaviz *Tool* existing class, an interactive object a user can grab to place at a particular position.

4.1.1 Designing the compass abstraction

First, we must declare a new *IA_Compass* interface that describes the features of our compass: these features are those inherited from the *IA_Tool* interface plus one new method.

```
package org.collaviz.iivc.abstraction ;
import ... ;

public interface IA_Compass extends IA_Tool {
   void setNorth (String target);
}
```

Then, we have to implement the *A_Compass* class that will inherit from the *A_Tool* class. As a compass must recompute its orientation each time it, its support or its target move, it must be declared as an object that observes its target (the observation of its support is ensured in the *A_Tool* class). This class must:

- declare a *String* for the name of its target, and a *Transform* to have an efficient access to the *Transform* of its target;
- implement a constructor according to some Collaviz constraints that registers the parameters used for its configuration and the new methods in order to be reached through its controller over the network;
- override some methods dedicated to the proper initialization, modification, and update of its instances;

- override other Collaviz methods to observe the modifications of the position of its target, and call the computation of the orientation if needed;
- implement a new method that computes the new orientation of the compass.

The declaration of a parameter and the new methods is written this way:

```
package org.collaviz.iivc.abstraction ;
import ... ;

public class A_Compass extends A_Tool implements
    IA_Compass {
   protected String northId ;
   protected IA_SupportedObject north;
   protected Transform targetTransform = null ;

   public A_Compass (String objectType, String objectName
       , IC_ObjectManager objectManager) {
      super (objectType, objectName, objectManager) ;
      parameters.put("North",northId);
      registerModificationCallback ("setNorth",
         new ICallbackHandler () {
            @Override
            public void callback (Object [] args) {
               setNorth((String)args[0]);
            }
         });
   }
```

The definition of a dependency to another Collaviz object must be done through a declaration in a parameters hashmap and through its registration as an observer of the other object:

```
   @Override
   public void setNorth (String target){
      northId = target ;
      parameters.put ("North", northId) ;
      north = (IA_SupportedObject)objectManager.getObject
            (northId) ;
      targetTransform = north.getTransform () ;
      north.registerDelayedObserver (id) ;
   }
```

The update and the modification of parameters are standardized in order to be automatically communicated to controller and presentation components:

```
   @Override
   protected void processUpdate (Map<String, Object>
       params) {
      super.processUpdate (params) ;
      final String _north = (String)params.get ("North");
      if (_north != null) {
         this.northId =_north ;
         parameters.put ("North",this.northId) ;
      }
   }

   @Override
   protected void processModify (Map<String, Object>
       params) {
      super.processModify (params) ;
      final String _north = (String)params.get ("North");
      if(_north != null)
         setNorth (_north) ;
   }
```

The observation of another Collaviz is made systematically through a dedicated method able to get the modifications and to inform the object (with a boolean) that modifications occurred:

```
   protected boolean needToCompute = false ;

   @Override
   public void delayedObservedPropertiesChanged (
      String obsName, Map<String, Object> chang) {
      super.delayedObservedPropertiesChanged (obsName,
          chang) ;
      if (supportObject != null &&
          supportObject.getId ().equals (obsName)) {
         final HashMap<String, Double> parentTransformMap
            = (HashMap<String, Double>)chang.get ("
               Transform") ;
         if (parentTransformMap != null) {
            needToCompute = true ;
         }
      }
      IA_SupportedObject tmpObj =
         (IA_SupportedObject)objectManager.getObject(
            northId);
      if (tmpObj != null && tmpObj.getId ().equals (
          obsName)) {
         final HashMap<String, Double> tmpObjTransformMap
            = (HashMap<String, Double>)chang.get ("
               Transform") ;
         if (tmpObjTransformMap != null) {
            needToCompute = true ;
         }
      }
   }
```

The functional behavior of the object must be executed when one of the observed objects changed: here the orientation of the compass should be computed each time the support of the compass moved or the target of the compass moved:

```
   @Override
   public synchronized void execute () {
      super.execute () ;
      if (needToCompute) {
         compute () ;
         modified () ;
         needToCompute = false ;
      }
   }
```

Last, we must code the behavior of the object: the effective computation of its orientation:

```
   private void compute () {
      Vector3d posCompass = new Vector3d () ;
      Vector3d posTarget = new Vector3d () ;
      Vector3d direction = new Vector3d () ;
      transform.getTranslation (posCompass) ;
      targetTransform.getTranslation (posTarget) ;
      direction.sub (posTarget, posCompass) ;
      Quat4d orientation = new Quat4d () ;
      Transform.getRotationBetween2Vectors (new
         Vector3d (0, 1, 0), direction, orientation) ;
      transform.setRotation (orientation) ;
      Map<String, Object> params = new HashMap<> () ;
      modifications.put ("Transform", transform.
          getMapForm ()) ;
   }
}
```

4.1.2 Designing the compass control

In this part, we must declare a new *IC_Compass* interface that describes the features of our compass: these features are those inherited from the *IA_Compass* and the *IC_Tool* interfaces.

```
package org.collaviz.iivc.control ;
```

DRAFT

```
import ... ;

public interface IC_Compass extends IA_Compass, IC_Tool{}
```

Then, we must declare a new *CService_Compass* class that inherits from the *C_Tool* class and that implements the new method of the compass, ensuring to redirect it to its associated abstraction.

```
package org.collaviz.iivc.control.service ;
import... ;

public class CService_Compass extends CService_Tool
                              implements IC_Compass {
   public CService_Compass (IA_SharedObject abstraction,
         boolean referentProxyArchi, int accessLevel,
         CService_ObjectManager objectManager) {
      super (abstraction, referentProxyArchi,
            accessLevel, objectManager) ;
   }
   @Override public void setNorth (){
      callModificationMethod ("setNorth");
   }
}
```

A very similar *CClient_Compass* must be defined. The only difference with the *CService_Compass* lays in one of its ancestors that propagates the messages on the network in a different way than the ancestor of the *CService_Compass*.

4.1.3 Designing the compass presentation

Last, we have to couple our compass with as many 3D graphics representations than 3D APIs usually used for the rendering of our VE. In this paper we will focus only on how to provide a 3D representation of the compass with Java3D.

Here we must describe the *IP_Compass* interface, which is independent from any 3D graphics API.

```
package org.collaviz.clientJava3D.pJava3D ;

public interface IP_Compass extends IP_Tool,
     ISceneGraphObject {
}
```

Then, we must implement the *PJava3D_Compass* class, which will update the transformation node of presentation component when its position or orientation are modified.

```
package org.collaviz.clientJava3D.pJava3D ;
import ... ;

public class PJava3D_Compass extends PJava3D_Tool
                             implements IP_Compass {
   public PJava3D_Compass (IC_SharedObject ctr,
         Vector3d translation, Quat4d rotation, Vector3d
             scale,
         PJava3D_ObjectManager presObjManager, String
             geometry){
      super (ctr, translation, rotation, scale, geometry,
            presObjManager) ;
   }


   @Override
   public void update (String userId, Map<String, Object>
         params, IC_SharedObject source) {
      super.update (userId, params, source) ;
      Map<String, Double> transformMap =
        (Map<String, Double>) params.get ("Transform") ;
      if (transformMap != null) {
         transform = new Transform (transformMap) ;
         updateTransform (transform) ;
      }
   }
}
```

4.2 Instantiating the virtual objects

Here we instantiated 3 objects in our VE: one compass, another virtual object that will act as its support, and a third virtual object that will be used as its target. We will also need a 3D cursor (which is an interaction tool) to select and move the compass, its support, or its target.

```
<virtualObject id="compass1" type="Compass">
   <owners>All</owners>
   <accessLevel>3</accessLevel>
   <refProxy>true</refProxy>
   <becomeReferent>true</becomeReferent>
   <becomeOwner>true</becomeOwner>
   <param name="Transform" type="Transform">
      0.0 0.0 0.0 0.0 0.0 0.0</param>
   <param name="Support" type="String">support</param>
   <param name="north" type="String">north</param>
</virtualObject>

<virtualObject id="support" type="Tool">
   <owners>All</owners>
   <accessLevel>3</accessLevel>
   <refProxy>true</refProxy>
   <becomeReferent>true</becomeReferent>
   <becomeOwner>true</becomeOwner>
   <param name="Transform" type="Transform">
      0.0 0.0 0.0 0.0 0.0 0.0</param>
</virtualObject>

<virtualObject id="north" type="SupportedObject">
   <owners>All</owners>
   <accessLevel>3</accessLevel>
   <refProxy>true</refProxy>
   <becomeReferent>true</becomeReferent>
   <becomeOwner>true</becomeOwner>
   <param name="Transform" type="Transform">
      0.0 10.0 0.0 0.0 0.0 0.0</param>
</virtualObject>

<virtualObject id="Hand1_User1" type="Cursor3D">
   <owners>All</owners>
   <accessLevel>3</accessLevel>
   <refProxy>true</refProxy>
   <becomeReferent>true</becomeReferent>
   <becomeOwner>true</becomeOwner>
   <param name="Offset" type="Transform">
      0 0 0 0 0 0 0.03 0.03 0.03</param>
   <param name="Color" type="Color">1.0 0.0 0.0</param>
   <param name="Geometry" type="String">cube</param>
</virtualObject>
```

4.3 Deploying a collaborative application

This part consists of placing these objects in the right description files, associated to either the server or some clients of the VE. When one object is declared on both the server and one client description files, the referent for this object migrates on the client.

Then, on each client, we must describe how the interaction tools, such as our 3D cursor, will be driven by an input device at run-time.

```
<input tool="Hand1_User1">
 <interactor id="mouse1">
  <event id="click" value="button1" action="pick" />
 <event id="unclick" value="button1" action="unpick" />
 <event id="moved" action="translate-world" />
 <event id="dragged" value="button1" action="translate"/>
 <event id="dragged" value="button2" action="translate"/>
```

DRAFT

```
 <event id="dragged" value="button3" action="translate"/>
  <event id="wheel" action="translate-z" />
 </interactor>
</input>
```

4.4 Limitations of this development process

As explained in this section, the Java classes to develop mix both the definition of the concept and some technical concerns mandatory for an integration in Collaviz. Collaviz experts noted that these classes are conceptually the same from one concept to another one. We highlighted 2 limitations of such a process:

L_1 the concept creator should focus on the design of concepts only;

L_2 the manual development of such Java classes is cumbersome and error-prone.

Moreover, the development task of world instantiators involve the creation of XML files corresponding to virtual worlds using the designed virtual concepts. We also highlighted 2 limitations of such a process:

L_3 the XML files must conform the concepts but no verification process exists for checking the conformance of XML files against the corresponding Java classes;

L_4 XML files are intrinsically verbose and not easy to read hindering their manual editing.

The following sections detail the benefits of using MDE for the development of CVEs.

5 Model Driven Engineering

The traditional way scientists use to master complexity is to resort to modeling. According to Jeff Rothenberg, "*Modeling, in the broadest sense, is the cost-effective use of something in place of something else for some cognitive purpose. It allows us to use something that is simpler, safer or cheaper than reality instead of reality for some purpose. A model represents reality for the given purpose; the model is an abstraction of reality in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, avoiding the complexity, danger and irreversibility of reality*" [18]. So modeling is not just about expressing a solution at a higher abstraction level than code. Modeling is indeed one of the touchstones of any scientific activity. In engineering, one wants to break down a complex system into as many models as needed in order to address all the relevant concerns in such a way that they become understandable enough [13].

The fundamental idea of Model-Driven Engineering (MDE) [19] is to consider models as first-class entities. The traditional organization followed in MDE is depicted in Figure 2. Each model conforms to a well-defined metamodel that describes the concepts and relationships of a given domain. Each metamodel conforms to a meta-metamodel, a self-described language for defining metamodels. A metamodel is the central part of a Domain-Specific Language (DSL). It defines the abstract syntax of the DSL. The Eclipse platform has become the de-facto back-end for defining and tooling DSL, notably through its modeling framework EMF[6]. Both open source and commercial tools leverage this support for developing tools dedicated to the development of DSLs and their editors. Sirius[7] is one of these tools that permits the generation of graphical editors using a metamodel and a model describing the graphical representation of the metamodel elements. Once the metamodel and some tools of a DSL created, models produced by using the DSL can be manipulated and transformed for various purposes using:

[6] `http://www.eclipse.org/modeling/`
[7] `http://www.eclipse.org/sirius/`

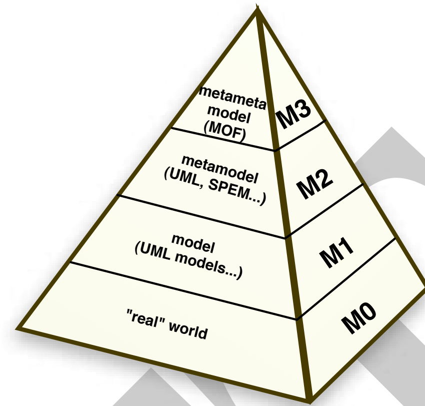

Figure 2: MDE organization

- *Model to Model transformations (M2M)*: A M2M transformation is a program that transforms an input model in an output model. The input and output models can have the same metamodel or different metamodels. There exist several model transformation languages, such as Kermeta [14].
- *Model to Text transformations (M2T)*: A M2T transformation is a program that transforms an input model in text (usually source code). It is possible to use any model transformation language to generate text.

6 MDE process for developing CVEs

MDE relies on 2 core concepts that tackle the limitations L_1 and L_2: the definition of domain specific languages (DSL) and model transformations (more precisely in our case: code generation). Instead of using Java for designing concepts, Collaviz experts would now define a DSL dedicated for this purpose. The benefits would be:

B_1 focusing on the definition of concepts only;

B_2 automatically generating the Java classes corresponding to the concept thanks to a model transformation.

Another core MDE principle is that models (virtual worlds in our case) produced using a DSL conform to it by construction, providing the following benefit:

B_3 no manual verification process is required to assure such a conformance, tackling therefore the limitation L_3.

The limitations L_2 and L_4 both concern a problem of usability in the editing process. Some MDE tools are dedicated to the creation of editing tools for DSLs. These tools use the metamodel of the DSL and a description of the wanted editor for generating, for instance, Eclipse plug-ins. Such editors can either be textual or graphical depending on the nature of the DSL. The benefit of such editors would be:

B_4 providing concept creators and world instantiators with dedicated editing tools.

In the next section, we detail how we technically used an MDE process for developing CVEs.

7 Leveraging MDE for the Collaviz framework: a first experiment

In this section, the MDE process applied to the Collaviz framework is described. Then, the benefits of the proposed process are discussed.

DRAFT

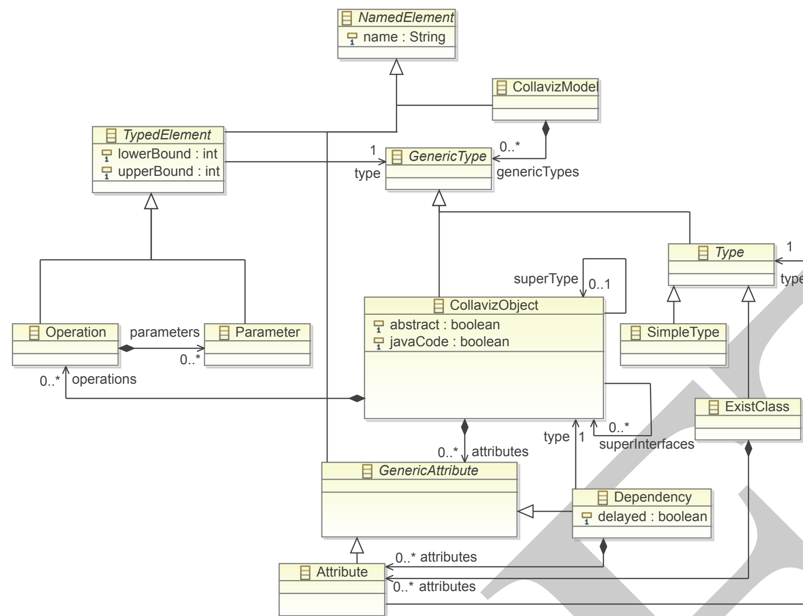

Figure 3: The Collaviz metamodel

7.1 Description of the applied MDE process

We applied an MDE process, depicted by Figure 4, for the Collaviz framework. This process tackles the limitations of the former process by leveraging MDE principles as detailed in the previous section. In opposite to the previous process, the proposed one smoothly combines the different actors of the framework.

The first column of Figure 4 focuses on tools for concept designers. As previously motivated, a DSL has been defined for helping concept creators in designing virtual concepts. The Eclipse Modeling Framework (EMF) provides tools for this purpose. The first step consists of defining a metamodel, depicted by Figure 3, that identifies Collaviz concepts and their relations. Such concepts are for examples a Collaviz object (class *CollavizObject*) that can have operations, attributes, a super type, and implement interfaces. A type (class *Type*) defines virtual objects that are not Collaviz ones. Both Collaviz and Type objects inherits from *GenericType* that defines the concept of an object type. A Collaviz model (class *CollavizModel*) is composed of *GenericType* instances. This metamodel conforms to the Ecore meta-metamodel, a metamodel for defining metamodels. The metamodel has been precised with constraints that check some structural properties.

Using this metamodel, an editor has been automatically generated as an Eclipse plug-in. Such an editor allows concept designers to create models of the Collaviz metamodel (a model being the set of concepts wanted by the designer). For instance, Figure 5 is a screen-shot of this editor during the definition of a Collaviz model. This model defines the previously introduced concept of compass. One can see that *Compass* is an instance of the class *CollavizObject* defined in the Collaviz metamodel (Figure 3). This *Compass* is composed of an operation *setNorth* and an attribute of type *Dependency* called *north*. The definition of the metamodel and its use to create models correspond to the benefits B_1 and B_3 (see Section 6) stating that: MDE permits developers to focus on the concepts of the domain under study; the conformance of a model is automatically checked against its metamodel.

From such a model is generated Java code dedicated to the Collaviz platform. This code generation is performed by a model transformation compiling model elements in Java code. The model-oriented programming language Kermeta has been used to develop this transformation within Eclipse. This step highlights the benefit B_2 on the use of model transformations to generate code and thus alleviate the coding effort.

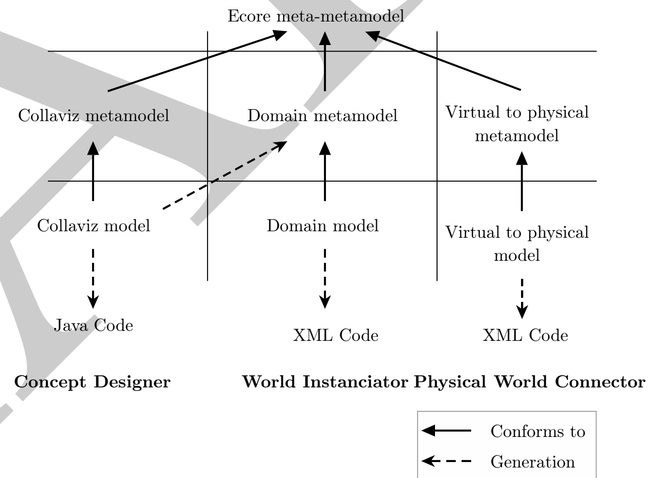

Figure 4: Process description

At this point, concept designers have tools for editing virtual concepts and generating their Java code. These concepts must now be used by world instantiators to create virtual worlds. For instance following the example of Figure 5, one may want to design a virtual world including one compass supported the hand of a user and targeting a landmark. To do so, a Collaviz model designed by concept designers *must be reified as a domain language* to be used by world instantiators. Another model transformation has been developed in Kermeta to automatically transform a Collaviz model in a domain metamodel.

DRAFT

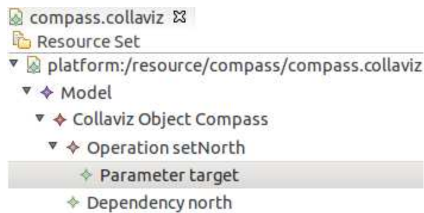

Figure 5: A Collaviz model, instance of the Collaviz metamodel depicted in Figure 3

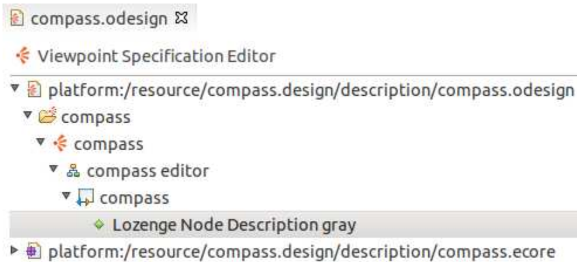

Figure 6: Defining the graphical representation with Sirius

Building scenes is a graphical job that needs a 2D or 3D graphical editor. Sirius is a model-driven tool dedicated to the creation of 2D graphical model editors. It requires the domain metamodel and a description of how the elements of this metamodel must be graphically represented. Figure 6 is a screenshot of the Sirius environment while defining a graphical representation. Each concept of the metamodel is mapped to a graphical representation. In our example, a compass is represented by a triangle (*Lozenge*). The graphical editors can also be customized. For instance, the node *Section* defines an editing toolbar. Once this last defined with Sirius by the concept designers, a 2D graphical editor can be generated as an Eclipse plug-in. This editor can be used by world instantiators to create virtual scenes (domain models) in a 2D environment, as illustrated in Figure 7. A third model transformation has been developed for generating the XML code, describing virtual scenes, that can read the Collaviz platform. The definition and generation of editors highlights the benefit B_4 on providing concept creators and world instantiators with dedicated editing tools.

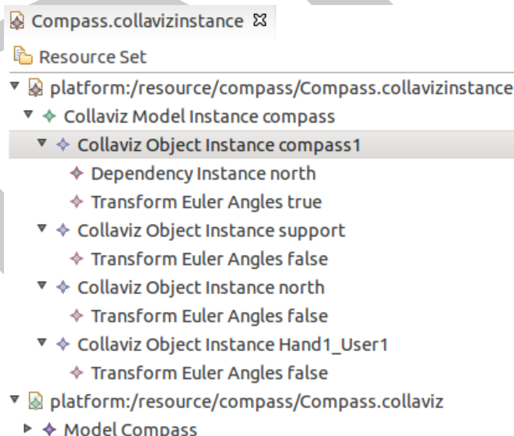

Figure 7: An example of Collaviz models instanciations

Finally, the physical world connector can bind the VE with some physical devices (the third column in Figure 4). Another metamodel has been defined for this purpose. Also, a fourth model transformation has been developed generating the XML code dedicated to configure these bindings.

7.2 Discussion about the benefits

7.2.1 Designing virtual objects

Concerning our small example with the compass, our solution generates a little bit more than 89% of the Java source code: it represents here a total of 186 lines from which 166 have been generated by our model transformations. It entirely generates the 3 interfaces, the 2 controller classes, and the main structure of the abstraction class (**only the content of the *compute* method, listed boldface in section 4.1.1, has to be coded manually**) and of the presentation class (**only the content of the *update* method, listed boldface in section 4.1.3, has to be coded manually**, and in practice here this step could have been avoided since already coded in one ancestor of this class).

The application described in [16], which deals with communication between an end-user and an ergonomist in order to enhance the design of an industrial workstation, has been built using the tools described in this paper. This is a "real" example of a new 3D application, with the addition to the Collaviz framework of 38 new Java interfaces and 48 new Java classes, which represents 8873 lines of Java code. In such a case our MDE approach generates 5338 lines of Java code, which represents 60% of the total amount of code.

Moreover, the designer can focus only on domain specific coding tasks (for the example, computing the orientation of the compass) without any concern neither for how to subscribe to the modifications of its support and target, nor for how to make it work on the network. The only other thing to be considered is how to update a presentation component using a specific 3D API, and this task could be given to another designer specialized in 3D graphics programming.

7.2.2 Instantiating virtual objects

Our solution generates the whole XML code describing the instantiations of our virtual objects. Moreover, it can check if this description is syntactically correct. For now these descriptions must be added manually in the main XML file describing a VE of in secondary XML files describing each user's own VE.

7.2.3 Deploying collaborative applications

This part has not been addressed yet, but here again, in a short future, we should be able to generate a great part of our configuration files for the setup of our CVE.

8 Conclusion

In this paper we explained the benefits of using MDE in the development process of CVEs. We illustrated our tooled proposal with a concrete use case. This use case shown how existing MDE tools can be used to create languages and their associated tools such as graphical editors. Such graphical editors are currently limited to 2D representations. Future work may focus on the generation of 3D editors thanks to dedicated frameworks such as GEF3D[8]. Besides, we applied our MDE process to the Collaviz framework. Adapting our approach to another VR or CVE framework is possible by developing new code generators but the process remains the same. Future work may focus on that to strengthen the validation.

Acknowledgment

We wish to thank Yorick Perret for his useful work on this project.

[8] `http://www.eclipse.org/gef3d/`

DRAFT

References

[1] P. Chevaillier, T.-H. Trinh, M. Barange, P. De Loor, F. Devillers, J. Soler, and R. Querrec. Semantic modeling of virtual environments using mascaret. In *Software Engineering and Architectures for Realtime Interactive Systems (SEARIS), 2011 4th Workshop on*, pages 1–8, 2011.

[2] C. Codella, R. Jalili, L. Koved, and J. Lewis. A toolkit for developing multi-user, distributed virtual environments. In *IEEE Virtual Reality Annual International Symposium*, 1993.

[3] D. Conner, S. Snibbe, K. Herndon, D. Robbins, R. Zeleznik, and A. van Dam. Three-dimensional widgets. In *SIGGRAPH : Symposium on Interactive Graphics*, pages 187–193, 1992.

[4] J. Coutaz. PAC: An Object Oriented Model for Implementing User Interfaces. *SIGCHI Bull.*, 19(2):37–41, 1987.

[5] 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 SIGGRAPH'93*, pages 135–142, New York, NY, USA, 1993. ACM.

[6] R. Dachselt, M. Hinz, and K. Meissner. Contigra: An xml-based architecture for component-oriented 3d applications. In *Proceedings of the Seventh International Conference on 3D Web Technology*, Web3D'02, pages 155–163, New York, NY, USA, 2002. ACM.

[7] T. Duval and C. Fleury. PAC-C3D: A New Software Architectural Model for Designing 3D Collaborative Virtual Environments. In *ICAT 2011*, pages 53–60, Osaka, Japan, Nov. 2011. VRSJ.

[8] P. Figueroa, M. Green, and H. J. Hoover. Intml: A description language for vr applications. In *Proceedings of the Seventh International Conference on 3D Web Technology*, Web3D '02, pages 53–58, New York, NY, USA, 2002. ACM.

[9] C. Fleury, T. Duval, V. Gouranton, and B. Arnaldi. A New Adaptive Data Distribution Model for Consistency Maintenance in Collaborative Virtual Environments. In *JVRC 2010 (2010 Joint Virtual Reality Conference of EuroVR - EGVE - VEC)*, pages 29–36, Fellbach, Germany, Sept. 2010.

[10] E. Frécon and M. Stenius. "DIVE : A scaleable network architecture for distributed virtual environments". *Distributed Systems Engineering*, 5:91–100, 1998.

[11] A. Goldberg. Information models, views, and controllers. *Dr. Dobb's J.*, 15:54–61, May 1990.

[12] J. Hartman and J. Wernecke. *The VRML 2.0 Handbook : Building Moving Worlds on the Web*. Addison-Wesley Publishing Company, 1996.

[13] J.-M. Jézéquel. Model Driven Design and Aspect Weaving. *Journal of Software and Systems Modeling (SoSyM)*, 7(2):209–218, 2008.

[14] J.-M. Jézéquel, B. Combemale, O. Barais, M. Monperrus, and F. Fouquet. Mashup of Meta-Languages and its Implementation in the Kermeta Language Workbench. *Software and Systems Modeling*, 2013.

[15] M. Lenk, A. Vitzthum, and B. Jung. Model-driven Iterative Development of 3D Web-applications Using SSIML, X3D and JavaScript. In *Proceedings of the 17th International Conference on 3D Web Technology*, Web3D '12, pages 161–169. ACM, 2012.

[16] C. Pontonnier, T. Duval, and G. Dumont. Sharing and bridging information in a collaborative virtual environment: application to ergonomics. In *IEEE international conference on cognitive infocommunication*, pages 121–126, Budapest, Hungary, Dec. 2013.

[17] T. Reenskaug. The original MVC reports. http://heim.ifi.uio.no/˜trygver/2007/MVC_Originals.pdf, 1979.

[18] J. Rothenberg. *The nature of modeling*, volume 3027. 1989.

[19] D. C. Schmidt. Guest editor's introduction: Model-driven engineering. *Computer*, 39(2):0025–31, 2006.

[20] A. Steed. "Some Useful Abstractions for Re-Usable Virtual Environment Platforms". In *Software Engineering and Architectures for Realtime Interactive Systems - SEARIS*, 2008.

[21] R. M. Taylor, II, T. C. Hudson, A. Seeger, H. Weber, J. Juliano, and A. T. Helser. VRPN: A Device-independent, Network-transparent VR Peripheral System. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, VRST '01, pages 55–61, New York, NY, USA, 2001. ACM.

[22] A. Vitzthum and B. Jung. Iterative Model Driven VR and AR Development with Round-Trip Engineering. In *SEARIS 2010 (IEEE VR 2010 Workshop on Software Engineering and Architectures for Realtime Interactive Systems)*, pages 3–8, Waltham, United States, Mar. 2010. Shaker-Verlag.

DRAFT