AGENT: Automatic Generation of Experimental Protocol Runtime

Gwendal Le Moulec, Ferran Argelaguet Sanz, Valérie Gouranton, Arnaud Blouin, Bruno Arnaldi

AGENT: Automatic Generation of Experimental Protocol Runtime

Gwendal Le Moulec* INSA Rennes, IRISA, Inria
Ferran Argelaguet† Inria, IRISA
Valérie Gouranton‡ INSA Rennes, IRISA, Inria, France
Arnaud Blouin§ INSA Rennes, IRISA, Inria
Bruno Arnaldi¶ INSA Rennes, IRISA, Inria

Abstract

Due to the nature of Virtual Reality (VR) research, conducting experiments in order to validate the researcher's hypotheses is a must. However, the development of such experiments is a tedious and time-consuming task. In this work, we propose to make this task easier, more intuitive and faster with a method able to describe and generate the most tedious components of VR experiments. The main objective is to let experiment designers focus on their core tasks: designing, conducting, and reporting experiments. To that end, we propose the use of Domain-Specific Languages (DSLs) to ease the description and generation of VR experiments. An analysis of published VR experiments is used to identify the main properties that characterize VR experiments. This allowed us to design AGENT (Automatic Generation of ExperimeNtal proTocol runtime), a DSL for specifying and generating experimental protocol runtimes. We demonstrated the feasibility of our approach by using AGENT on two experiments published in the VRST'16 proceedings.

1 Introduction

One of the focus of Virtual Reality (VR) researchers is the analysis of how humans behave [27], perceive [15] and interact [2] in VR environments. This research is bounded to the design of user studies in order to validate the researcher's hypotheses. In particular, experiments in VR aim at studying the effects of a system, application, interface, algorithm, *etc.* on system or users. Indeed, according to Andrew Colman [4], "experimental methods [...] [allow] rigorous examination of causal effects". For example, Latoschick *et al.* [17] designed a fake-mirror system that consists of a screen displaying the picture of an avatar imitating the movements of the user. They conducted an experiment to study the effect of the avatar nature (more or less realistic) on the feeling the users had they were standing before a real mirror.

Experiments must conform to some requirements, such as defining experimental conditions, dependent and independent variables, and the experimental protocol [8]. Research works were conducted to: guide experiment design in VR and augmented reality [10, 16, 14]; help evaluating specific concepts such as presence [25, 28], acceptability [3], or collaboration [13, 20]. Yet, experiment designers still have to develop the experimental VR application, which is a tedious, repetitive, and time-consuming task.

*e-mail: gwendal.le-moulec@irisa.fr
†e-mail: fernando.argelaguet_sanz@inria.fr
‡e-mail: valerie.gouranton@irisa.fr
§e-mail: arnaud.blouin@irisa.fr
¶e-mail: bruno.arnaldi@irisa.fr

In this work, we propose AGENT (Automatic Generation of ExperimeNtal proTocol runtime) to ease the development of experimental VR applications. AGENT automatically generates experimental protocol runtimes letting experiment designers focus on their core tasks: designing, conducting, and reporting experiments. AGENT is based on the increasingly used Software Engineering concept of Domain-Specific Language (DSL) [31, 9, 21]. DSLs ease software production by using software languages designed to tackle specific problems. DSLs are indeed designed to have key-words, notations, and syntaxes familiar to the domain experts. DSLs should remain small, simple, and easy to use. Their design should be adapted to the organizational process of the final users [32]. By fulfilling these criteria, DSLs allow to leverage specific domain expertise of various stakeholders involved in the development of software systems [10, 16, 14].

After an experimental protocol designed within an AGENT model, this last is compiled into runnable code. The runnable code is then integrated into an already existing VR project, provided by the experiment designer.

The paper is structured as follows. Section 2 presents the state of the art of VR experiment design, the efforts made to ease the task of experiment designers, and their limits. In Section 3, an analysis of several existing and reported VR experiments is presented to characterize VR experiments. Based on this analysis, the proposed approach is detailed in Section 4. Section 5 reports and discusses the usage of our approach on two use cases. Finally, Section 6 concludes this work and presents future works.

2 State of the Art

2.1 Basics of Experimentation

Designing experiments is based on a rigorous set of principles to follow [8]. According to Andrew Colman [4], an experiment is "*a research method whose defining features are manipulation of an independent variable or variables and control of extraneous variables that might influence the dependent variable*". An independent variable is *"a variable that is varied by the experimenter independently of the extraneous variables"*. A dependent variable is *"a variable that is potentially liable to be influenced by one or more independent variables"*. The main idea behind experiments is to assess what causality relations exist between different events or facts [4], *i.e.,* the potential effect independent variables have on dependent variables. Dependent variables correspond to the data collected during experiments and can be qualitative or quantitative data [5]. Asserting the existence of an apparent causal relation between independent and dependent variables, *i.e.,* ensure internal validity, is a necessary step [26]. Generalizing results observed on small populations, *i.e.,* ensure external validity, requires the use of statistical methods [23, 7].

2.2 VR Experiment Design

Those general principles are applicable in every domain where experiments are used. In Computer Science, domains related to human-centered design need experiments to validate their approaches. Human-centered design indeed often implies human-computer interactions: the influence of the design properties on user-experience should be validated. Gabbbard proposed guidelines and methods for designing human-centered applications and experiments [10]. Other works focus on augmented reality applications design and evaluation [16, 14].

If the classic method consists in running experiments where a population of participants uses the interfaces to evaluate, other approaches have been proposed. Stanney *et al.* use heuristics to guide and evaluate VR interfaces design [29]. Tromp *et al.* propose a usability inspection method [30], *i.e.,* a simulation of the use of an application to find users need.

In human-centered design and more specifically in VR, some specific interface characteristics are evaluated. They are often qualitative aspects, which are not easy to evaluate because of their subjectivity [8]. Research have been done to propose standard questionnaires. Congnitive loading induced by systems can be evaluated thanks to the NASA-TLX questionnaire [12]. The Situation Present Assessment Method (SPAM) can be used to evaluate presence [6], so as other methods [25, 28, 33]. Brooke proposed an acceptability questionnaire [3]. Evaluation of collaboration was studied too: Hornbæk proposed metrics based on communication (*e.g.,* number of uttered words per person, number of questions asked to collaborators, number of interruptions) [13]. Meier *et al.* completed these metrics by considering other aspects of collaboration (*e.g.,* coordination or motivation of each actor) [20].

2.3 Easing Experiment Design

If these works focus on guiding experiment designers, implementing experimental protocols to be integrated into running applications is a manual, time-consuming and tedious task. Tools dedicated to facilitate this task exist. Field *et al.* proposed IBM SPSS Statistics [7], a tool for performing statistical analyses of data. Another example of such tools is the R project[1]. Software solutions for designing experimental conditions (*e.g.,* variables, populations) and registering results can be used, *e.g.,* EDA[2] or Go-Lab[3], but they are mostly useful for other domains than VR (mostly biology, physics, or chemistry). VR could benefit of some tools referenced by the National Heritage Language Resource Center (NHLRC, California)[4] that are more human-centered. More interestingly, the framework EVE was specifically designed to ease experiment setup and implementation in Virtual Environments [11]. EVE focuses on automating data gathering and analysis but does not handle experimental protocol generation. No other solutions dedicated to VR exist.

In overall, the existing solutions are limited to experiment conditions modeling and data management. There is no code generation and the model are in general not meant to be processed by programs. Furthermore, protocol definition is often limited or nonexistent. The consequence we observed ourselves is that researchers end-up by developing their own limited and *ad hoc* solutions. There are no libraries or projects that generate running code for VR experiments.

[1]https://www.r-project.org/

[2]https://www.nc3rs.org.uk/experimental-design-assistant-eda

[3]http://www.golabz.eu/apps/experiment-design-tool

[4]http://international.ucla.edu/nhlrc/data/software

2.4 Model-Driven Engineering and Domain-Specific Languages

Software industry faces the constant increase of systems complexity. Modelling aims at mastering this complexity through the Model-Driven Engineering (MDE) domain [24]. In MDE, models focus on specific problems for a specific audience to ease the software development process. MDE tools help software engineers in developing and tooling languages that are designed to answer specific problems; such languages are called domain-specific languages (DSL) [31, 9, 21]. DSLs provide a bridge between the problem space in which domain experts (experiment designers in our case) work and the implementation space. Developed DSLs usually come with associated tools such as editors, code generators, simulators.

3 VR Evaluations: a Preliminary Analysis

In this section we study recent research papers to draw up a precise map of the concepts related to VR experiment design. The goal is to identify the properties that characterize experiments in VR.

We have studied 15 papers and posters from the VRST'16 proceedings [1] where experiments are conducted on various topics: displays, latency, training, presence, human behavior simulation, haptics, tracking, cinematic VR, distance perception, and 3D user interfaces. We focused on experiment design: mathematical and statistical analysis of data is not in the scope of this paper (but in future works). Obviously, the set of identified properties - obtained empirically - can not cover all VR experiments. Though, we consider that recent VRST papers are representative enough. Hence, the drawn properties cover a large scope of VR experiments.

The first observation we made is that designing an experiment can be divided into two main tasks: (1) experimental conditions and variable design, and (2) protocol design. We reported our analysis on two mind-maps, respectively for the concept of variable and the concept of protocol (see Figures 1 and 2). This allowed us to quickly obtain a structure of concepts specific to VR experiments. Figures 1 and 2 summarizes the main concepts we identified. Section 3.1 discusses experimental conditions and variable design. Section 3.2 discusses protocol design.

3.1 Variables

Two types of variables exist: dependent and independent variables. Dependent variables can be quantitative: physiological constants, *e.g.,* blood pressure, time for performing a task, performance of the subject, accuracy of a method. They can also be subjective feelings, *e.g.,* how the subject appreciated a task, how comfortable it was. Questionnaires like the ones presented in Section 2 can be used to record these qualitative measures. Hence, data can come from various sources: mainly physiological sensors, software measurements, and forms. As a result, two first properties can be drawn up to characterize VR experiments:

P1: dependent variables can be of different types (integer, float, boolean, mark, customized types, *etc.*)

P2: dependent variables correspond to three types of data sources: sensors, software measurements, and forms.

Independent variables correspond to what is under evaluation or comparison: metaphors, navigation, interaction

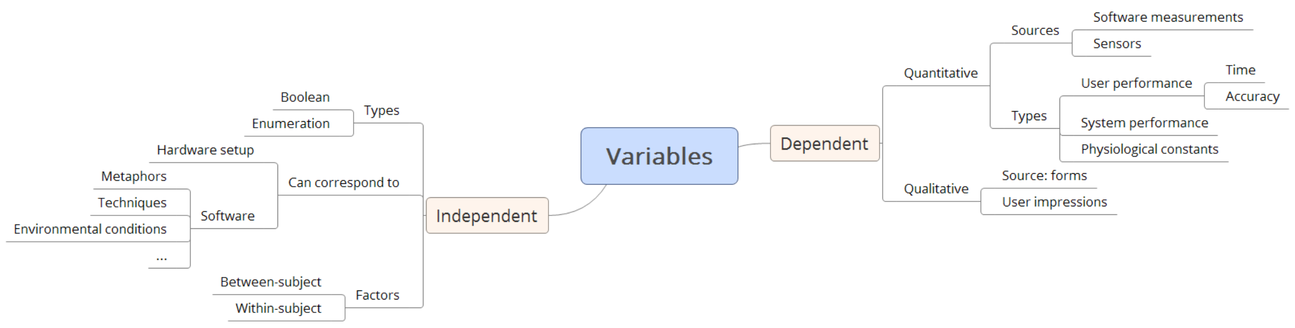


Figure 1: mind-map depicting the concept hierarchy under the concept of variables in VR experiments.

or rendering techniques, hardware setups, algorithms, environmental conditions, *etc.* Hence, independent variables correspond to software or hardware features developed or studied by the experiment designers. In experiments, two usages are possible for independent variables:

- comparison, *e.g.,* several metaphors are compared to determine which one is the most appreciated,
- determining the effect of a single condition, *e.g.,* one specific environmental condition is activated and then deactivated to determine the effect induced by its presence.

Independent variables can then be of two types: boolean (compare presence / absence) and enumeration (comparison of several conditions). The possible values of independent variables are called "levels". Two more properties can then be added:

P3: two types are possible for independent variables: boolean and enumeration.

P4: independent variables and their levels correspond to software or hardware features that can vary.

All possible levels are not necessarily presented to each participant of an experiment. In some cases, several groups of participants - exposed to different conditions - are necessary. For example, to evaluate the effect of a navigation technique on cybersickness, two groups could be made: one group of subjects exposed to the navigation technique under evaluation and one control group, exposed to a known navigation technique that does not induce abnormal cybersickness. Conditions that are evaluated on all participants of a study are called "within-subject factors", whereas conditions that differ from one group to another are called "between-subject factors". One more property can be drawn up:

P5: between-subject factors imply to make several groups, each of them corresponding to a distinct protocol.

3.2 Protocol

The protocol is the process that participants follow during a session. We consider that it begins when the participants start to use the VR experimental application, *i.e.,* we consider the protocol starts after preliminary phases during which the participants have to read and sign a consent form and are associated to identifiers, for anonymity reasons. If there are between-subject factors and hence several groups, each group is associated to a different protocol, with the only difference being at the level of the between-subject factors. This is the statement of property **P6**:

P6: the protocols of each group differ only at the level of between-subject factors.

The part of the protocol on which we focus is often separated into two phases: (1) an acclimatization and / or calibration phase and (2) a data acquisition phase. The acclimatization phase role is to train the participants to the use of the VR application: they should be used to the different conditions they will experiment. During the acclimatization phase, participants repeat several times the different conditions, very often in a randomized order. During this phase, it is possible to adapt the number of repetitions to the participants. A calibration phase with similar characteristics can be performed, *e.g.,* if some sensors must be calibrated.

The data acquisition phase role is to record data, and hence to evaluate the influence of the independent variables on the dependent variables. Participants repeat also several times the different conditions in a randomized order. The number of repetitions is generally greater than in acclimatization phase and must be the same for each participant. A last property can be added:

P7: in acclimatization and calibration phases, the subject perform tasks but no data is collected.

4 Approach

The properties that characterized VR experiments are now used to propose an approach for easing their design and production.

4.1 Overview

The proposed approach allows to design VR experiments with the use of a DSL we designed based on the properties identified in Section 3. This DSL is called AGENT (Automatic Generation of ExperimeNtal proTocol runtime). Models produced with this DSL are compiled into code to be integrated into VR projects (*e.g.,* Unity 3D projects). Figure 3 depicts the processing chain of the approach.

The use of AGENT produces an *AGENT model* that is then compiled into *runnable code* integrable into a VR project. An *AGENT model* is composed of two parts: an *experimental conditions model* and a *protocol model*. It is up to the experiment designer to provide to AGENT all the VR

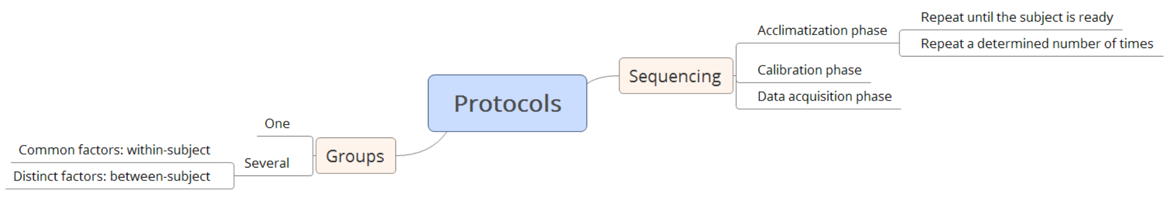

Protocols
Sequencing
Acclimatization phase
Repeat until the subject is ready
Repeat a determined number of times
Calibration phase
Data acquisition phase
Groups
One
Several
Common factors: within-subject
Distinct factors: between-subject

Figure 2: mind-map depicting the concept hierarchy under the concept of protocols in VR experiments.

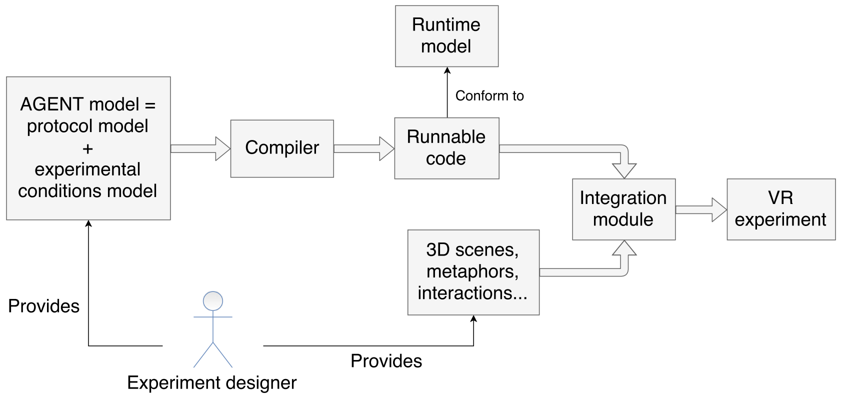


Figure 3: processing chain of the approach.

components that are not directly related to the experiment (*i.e.*, 3D models, metaphors, interactions, *etc.*).

The remaining of Section 4 is organized as follows. Section 4.2 introduces an illustrative example as a basis for the detailed explanation of the approach. Section 4.3 presents the *experimental conditions model* structure. Section 4.4 presents the *protocol model* structure. Section 4.5 ends the presentation of the approach by presenting the compilation and integration steps.

4.2 Illustrative Example

Consider a VR experiment of which independent variables are:

- I_b, a boolean variable (between-subject factor),
- I_e with two levels : L_1 and L_2 (within-subject factor).

The dependent variables are:

- D_q, answers to a questionnaire, in the form of Likert-scale marks, for evaluating the task regarding independent variables possible values,
- D_s, speed of execution of the task.

The protocol is as follows:

1. acclimatization phase : the subject executes the task 2×4 times, *i.e.,* 4 times each condition among the condition set C_b or $C_{\neg b}$, depending on the group (see Equations (1) and (2)), no data being recorded,
2. data acquisition : the subject executes the task 2×32 times, *i.e.,* 32 times each condition among C_b or $C_{\neg b}$, with D_s being recorded,
3. subjective evaluation : the subject answers the questionnaire.

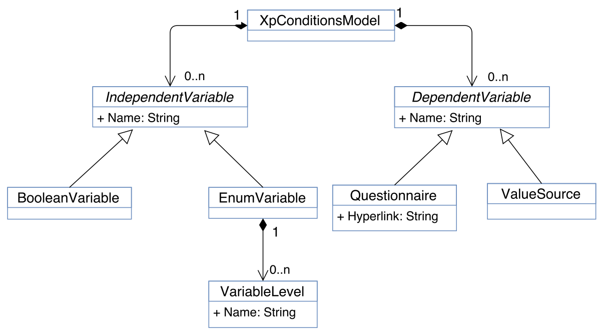


Figure 4: *experimental conditions model* UML class diagram.

$$C_b = \{(I_b, L_1), (I_b, L_2)\} \tag{1}$$

$$C_{\neg b} = \{(\neg I_b, L_1), (\neg I_b, L_2)\} \tag{2}$$

The two groups, respectively exposed to the conditions C_b and $C_{\neg b}$ are called G_b and $G_{\neg b}$.

4.3 Experimental Conditions Model

The *experimental conditions model* is the part of an *AGENT model* that describes the independent and dependent variables. Figure 4 depicts the concepts of this DSL in the form of a UML class diagram. Figure 5 shows the *experimental conditions model* designed using AGENT that corresponds to the example of Section 4.2.

An *experimental conditions model* has a tree-like structure. The tree is composed of three mandatory nodes: (1) the root that defines the name of the model, and its two child nodes; (2) the "independent variables" node; (3) the "dependent variables" node. The independent and dependent variables are respectively to be defined under the nodes (2) and (3), as their child nodes.

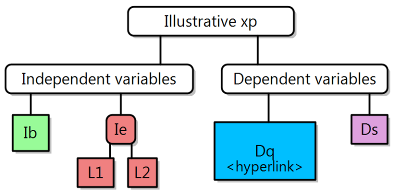


Figure 5: *experimental conditions model* based on the example of Section 4.2.

According to **P3**, there are two types of independent variables: boolean and enumeration variables. Enumeration variables are constituted of possible levels, each level being defined under the node corresponding to their variable (*e.g.,* see Figure 5).

There are two types of dependent variables: objective and subjective variables. Objective variables are data collected from software or hardware sources (physiological sensors and software measurements of **P2**) and can be of various types (*e.g.,* boolean, integer, enumeration, float, customized types, *etc.*), according to **P1**. The linkage of objective dependent variables to their data source is managed by the *integration module* (see Section 4.5), not by the *experimental conditions model*, where only the names of the variables can be provided (*e.g.,* D_s). Subjective variables are questions asked to participants and their answers, gathered into special forms designed by the experimenters (see **P2**). In the *experimental conditions model*, the experiment designer can define forms with an identifier (*e.g.,* D_q) and a hyperlink allowing to access the form. In the remaining of the paper, all the leaves of an *experimental conditions model* (*i.e.,* boolean independent variables, variable levels, questionnaires, and objective dependent variables) will be called *features*.

4.4 Protocol Model

4.4.1 Description

Figure 6 depicts the concepts that are presents in *protocol models* in the form of a class UML diagram. It shows that AGENT allows to define experimental protocols as lists. Figure 7 shows such a list, that contains three elements separated by arrows. Some elements are composed, *e.g.,* the second element in Figure 7 is composed of *Condition1*, *Condition2*, and *Acclim*. Figure 7 shows the *protocol model* designed using AGENT that corresponds to the group G_b of the example presented in Section 4.2. The protocol of group $G_{\neg b}$ is the same, but *feature* I_b is not present (set to false). Note that between-subject and within-subject factors are implicitly defined in AGENT. The distinction is made if several groups (hence several *protocol models*) exist: variables corresponding to between-subject factors are the ones that vary from a *protocol model* to another. Properties **P5** and **P6** are then satisfied.

A *protocol model* is a states list with five types of state: (1) start state (green disk on Figure 7), end state (red disk), simple state (yellow rectangle), random-loop state (purple rectangle), and customized-loop state (orange rectangle, see Figure 11). The core idea behind *protocol models* is that each phase of the experiment (simple, random-loop, and customized-loop states) corresponds to the selection of a subset of the *features* defined by an *experimental conditions model*. Let's consider the protocol given the example of Section 4.2. First, the subject goes through an acclimatization phase where he has to perform the task 8 times, covering 2 conditions that are combinations of the independent variables possible values. Second, he does the same thing but with 64 repetitions, and with data being recorded, *i.e.,* considering the effect of the independent variables on the dependent variable D_s. Third, he completes the questionnaire corresponding to the set of dependent variables identified by D_q. Hence, each step corresponds to the selection of several *features* from the *experimental conditions model* of Figure 5. For each step of a *protocol model*, the selected *features* are listed under the "Features" label (see Figure 7). For example, in the simple state *SubjectiveEval*, there is only one *feature* selected: D_q.

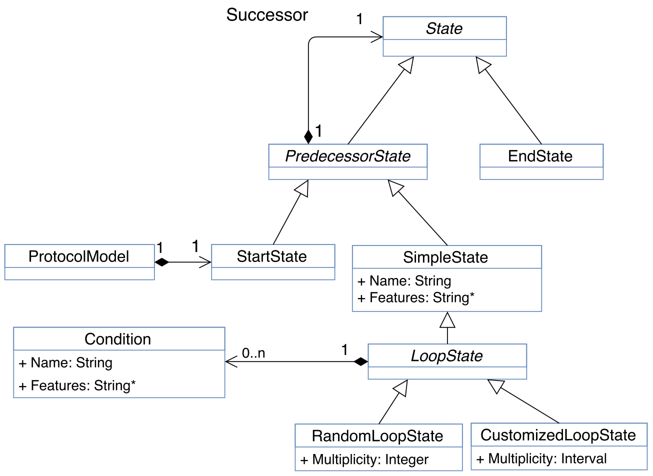


Figure 6: *protocol model* UML class diagram.

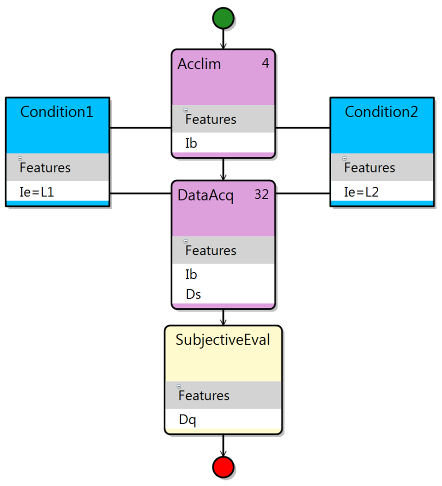


Figure 7: *protocol model* of group G_b, from the example presented in Section 4.2.

The case of loop states (random-loop and customized-loop) is more complex. A loop state models the repetition of a task under different conditions. The number of repetitions for each condition, *i.e.,* the multiplicity (see Figure 6), is indicated at the top-right corner of each loop state (see Figure 7). Conditions are modeled as blue rectangles linked to the loop state that cover them. Conditions make references to *features*, so as loop states do. If a *feature* is held by the loop state itself, then it means that this *feature* is active for all repetitions and conditions. For example, in the *DataAcq* state corresponding to the step (2) of the illustrative protocol, data is recorded whatever the current condition. The between-subject factor I_b is also set to true for the group G_b in all cases. Hence, the *feature* D_s is held by the random-loop state (as so as I_b for G_b). If a *feature* is held by a condition, then it means that the *feature* is active only for repetitions where the condition is active.

Random-loop states allow to model repetitions where conditions come in a random order. The multiplicity is an integer (see Figure 6) that represents the number of times each condition will be repeated. Customized-loop states allow to model other kinds of repetitions, *e.g.,* deterministic or based on the participants choice. Sometimes experiment designers indeed propose phases where participants can repeat conditions on demand [19]. To handle this case, the multiplicity of customized-loop states is not an integer but an interval (see Figure 6). This way, the experiment designer can make the number of loops be: constant (notation "n"), limited (notation "$n..N$"), or unlimited (notation "$n..*$").

Note that differentiating acclimatization and data acquisition phases is illustrated here: acclimatization phases are loop states where data recording is deactivated (no dependent variable in the *feature* list of the loop state). Data acquisition phases are on the contrary loop states (in general random-loop states) where data recording is activated. In our approach, acclimatization and calibration phases are represented the same way: extraneous calibrations of any kinds are not managed by AGENT. **P7** is then satisfied.

4.4.2 Discussion

The choice of modeling protocols as lists comes from the preliminary study (see Section 3). Lists are sufficient because of the nature of VR experiments: protocols do not allow alternatives. However, when there are between-subject variables, the protocol varies from one group to another. In our approach, the experiment designer simply has to make one *protocol model* per group. The only variations between the different protocols are at the level of the *features* corresponding to between-subject variables. That is why the concepts of between-subject and within-subject variables do not appear explicitly in AGENT.

Other variations may appear at the task level. The subject could indeed have the choice to execute some actions in the order he wants. For example, consider a task consisting in selecting multiple objects in the Virtual Environment, the subject could chose in which order he selects the objects. However, these variations are at the level of the use-case and do not correspond to variations of the protocol in itself. Variations in the use-case are out of the scope of this paper and that is why AGENT does not manage them. It is up to the experiment designer to manage these variations and provide them to AGENT along with the VR elements not related to the experiment in itself (see Figure 3).

4.5 Code Generation and Integration

After an experiment protocol was modeled, *runnable code* can be generated through the use of the AGENT *compiler*. The generated *runnable code* can be integrated into the VR project through the AGENT *integration module*. The *runnable code* is characterized in Section 4.5.1. The transformation operations that allow to compile *AGENT models* to *runnable code* are precised in Section 4.5.2. The *integration module* is presented in Section 4.5.3.

4.5.1 Generated Runnable Code Characterization

As Figure 3 shows, the generated *runnable code* can be characterized by a *runtime model*. The *runtime model* is a state machine. For example, the *runtime model* generated from the example of Section 4.2 (group G_b) is represented in Figure 8.

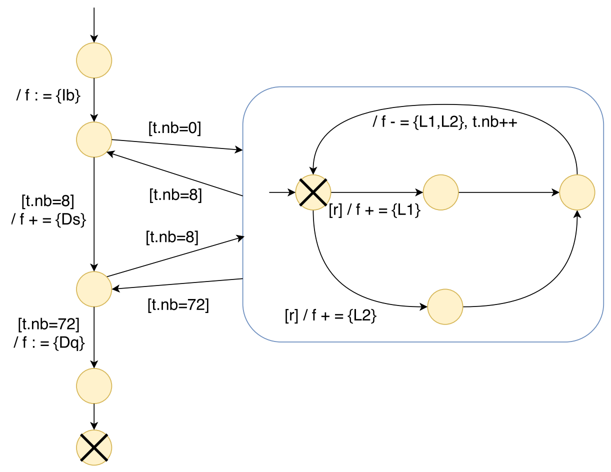


Figure 8: *runtime model* generated from the example of Section 4.2 (group G_b).

On the figure, The rounded rectangle is a state containing itself a state machine. Conditions triggering a transition are indicated between brackets. Actions resulting from the triggering of a transition follow the slash. The variable $t.nb$ is the number of executed trials. The variable f is the set of selected features. The $[r]$ condition means that the transition is triggered randomly. Final states are marked by a cross.

4.5.2 Compilation: from AGENT Model to Runnable Code

Figure 8 already gives the intuition of the transformation algorithm (from *protocol model* to *runtime model*). The formal algorithm will not be given for the sake of simplicity and concision. We will only give the intuition of it.

Transformation of Start, End, and Simple States The transformation operator applied to the start, end, and simple states is the identity: they are respectively transformed to initial, final, and standard states of the *runtime model*.

Transformation of Transitions Two possibilities exist:

1. the origin state of the transition is a start, end, or simple state. In that case, the transformation operator is the identity.

2. the origin state of the transition is a loop state. In that case, the generated transition is conditional: the condition for going through the generated transition is that the number of loops defined in the loop state where ran.

In all cases, the generated transitions must trigger events: the deselection of the features held by the origin state and the selection of the features held by the target state.

Transformation of Loop States Loop states are transformed to sub-state-machines. Each condition is transformed to a state. The sub-state-machine must loop on each of these states a number of time conform to the multiplicity of the loop state. Obviously features held by the conditions must be selected or deselected adequately.

4.5.3 Code Integration

The *integration module* is composed of 3 libraries that the experiment designer has to use in order to integrate properly the generated *runnable code* to the VR project. This section presents conceptually these libraries. Concrete usage of them is presented in Section 5.

Feature Binding Library The experiment designer must bind the *experimental conditions model features* to runtime features (*e.g.,* metaphors, interactions, virtual objects, *etc.*), that he provides (see Figure 3). For example, the *feature* L_1 represented in Figure 5 could be bound to a virtual object (runtime feature). The selection (resp. deselection) of L_1 in the *runtime model* would make the virtual object appear (resp.disappear). Conceptually, the *feature binding library* is a map that links *experimental conditions model features* to runtime features, with two functions to implement for specifying what happens at each selection / deselection of a feature. Objective dependent variables are associated to their data source (*e.g.,* a field in a class, the output of a measurement tool, *etc.*), that can be of various types. Property **P1**, **P2**, and **P4** are then satisfied.

Condition Sequencing Library This library provides several algorithms for sequencing the conditions in the case of loop states: random with constant seed, random with time-based seed, deterministic with fixed number of loops, controlled by the user, *etc.*.

Trial Completion Management Library Trial completion is managed by the *runtime model* at the experiment level, *i.e.,* loop conditions are changed after each trial completion. However, trial completion must be detected and the resulting effects on the Virtual Environment must be triggered. Consider for example a task consisting in going from a departure point D to an arrival point A. Trial completion should be detected when the arrival point A is reached and the triggered effect would be to make the avatar automatically return to the departure point D, for starting a new trial with other conditions. The experiment designer can use the *trial completion management library* to add conditions and effects corresponding to trial completion to the transitions of the *runtime model*.

4.6 Properties Fulfilling

Our approach satisfies the properties listed in Section 3.

- **P1**: the type of an *experimental conditions model feature* is the type of its bound runtime feature, which is freely determined by the experiment designer (see Section 4.5.3).
- **P2**: *Questionnaire* dependent variables are bound to forms thanks to an hyperlink (see Figures 4 and 5). *ValueSource* dependent variables are bound to data sources which nature is determined by the experiment designer.
- **P3**: the boolean and enumeration types are defined (Figure 4).
- **P4**: binding *experimental conditions model features* to runtime features is ensured (see Section 4.5.3).
- **P5** and **P6**: between and within subject factors are implicitly defined (see Section 4.4).
- **P7**: acclimatization phases are loop states where data recording is deactivated (see Section 4.4).

4.7 Implementation

AGENT was developed using *DSL Tools*[5], a Visual Studio 2015[6] extension for creating DSLs. The AGENT compiler has been developed in C#. Once an AGENT model designed by an experiment designer, code can be automatically generated from this model by the compiler. The generated code consists of XML files that implement the state machine structure of the *runtime model*. It comes along with pre-coded C# classes and an interpreter, provided with AGENT, that implements the dynamic aspect of the *runtime model*. The generated classes and the interpreter are meant to be used with Unity 3D (we discuss adaptation of AGENT for other VR platforms in Section 5.3). The integration module is a Unity library, presented in more details in Section 5.1.

5 Creating an Experiment Using AGENT

In this section, we explain step by step the usage of AGENT to produce an experiment. Section 5.2 presents the use of AGENT on two real use-cases reported in the ACM VRST 2016 proceedings [1]. We end this section with a discussion (Section 5.3).

5.1 Usage

The usage of AGENT is composed of three main steps: modeling, code generation, and code integration.

5.1.1 Modeling

In our implementation, the *experimental conditions model* and the *protocol model* are made using AGENT, developed as Visual Studio extensions. To produce one of the models, the experiment designer has first to create a new Visual Studio project with the appropriate model type. He can then create the model by drag-and-dropping the different components (*e.g., experimental conditions model features*, *protocol model* states, transitions, *etc.*) on a conception area. The text fields and *features* lists can be edited directly on the created components. Figures 5, 7, 10 and 11 show examples (screen captures) of models built using the AGENT DSL within Visual Studio. Once the models are conceived they are saved into XML files that are the input data for the code generation step.

5.1.2 Code Generation

The *compiler* takes as entry the XML files and generates an output XML file that represents the structure of the state machine. The output XML makes reference to C# classes we pre-coded and that are provided with AGENT. These classes are responsible of transitions implementation.

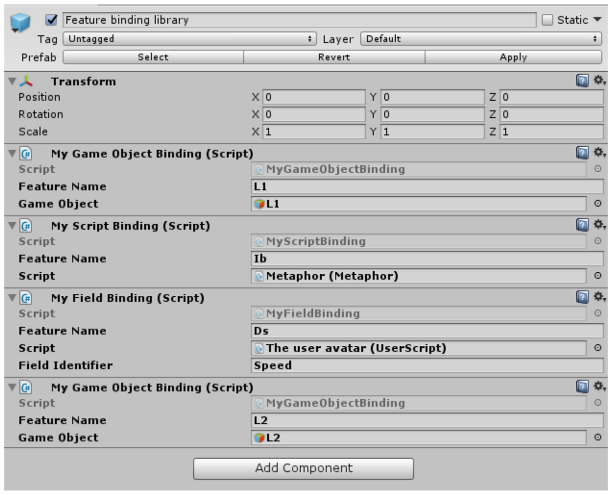
5.1.3 Code Integration

All the integration is done in Unity at the level of one provided prefab with three child nodes: one for each library defined in Section 4.5.3. They are each composed of Unity scripts that have to be inherited to produce new scripts to attach to the nodes of the provided prefab.

Feature Binding Library Figure 9 shows an example of feature binding in Unity, based on the example of Section 4.2. This library contains several scripts, each of them allowing to bind a feature to a special kind of Unity resource (*e.g.,* GameObject, script, field, *etc.*). Each of these script define at least two fields: one for specifying the *feature* name, and another for specifying the Unity resource it is bound

[5]`https://msdn.microsoft.com/en-us/library/bb126259.aspx`

[6]`https://www.visualstudio.com`



```
Feature binding library                         Static
Tag  Untagged          Layer  Default
Prefab   Select   Revert   Apply
Transform
Position    X 0   Y 0   Z 0
Rotation    X 0   Y 0   Z 0
Scale       X 1   Y 1   Z 1
My Game Object Binding (Script)
Script          MyGameObjectBinding
Feature Name    L1
Game Object     L1
My Script Binding (Script)
Script          MyScriptBinding
Feature Name    Ib
Script          Metaphor (Metaphor)
My Field Binding (Script)
Script            MyFieldBinding
Feature Name      Ds
Script            The user avatar (UserScript)
Field Identifier  Speed
My Game Object Binding (Script)
Script          MyGameObjectBinding
Feature Name    L2
Game Object     L2
Add Component
```

Figure 9: Unity *feature binding library* node attached scripts, based on the example of Section 4.2.

to. They also define two methods to be implemented by the experiment designer, for managing *feature* selection and deselection. To bind *features* to external data sources (*e.g.*, physiological sensors), the experiment designer can bind the desired *features* to a Unity script which reads the external data.

Condition Sequencing Library This library is composed of two scripts, corresponding to the two types of loop states. They allow to associate a loop state to sequencing algorithms we provide. The customized-loop script allows in particular to detect user requests to either end the loop or switch conditions.

Trial Completion Management Library This library is composed of one script that defines two functions to implement, for specifying the trial completion condition and the trial completion effect. A field allows to reference the *protocol model* state it is bound to.

5.2 Use-cases

We used our approach on two experiments. The first use-case is an experiment we reported [18]. The *AGENT model* was made, the code was generated, and then integrated to the Unity project of the associated VR application. (obviously, the former code parts that were responsible of the protocol runtime were removed). The refactored experiment is fully functional.

The second use-case was the experiment reported by Mossel *et al.* [22]. In this experiment, Mossel *et al.* compare two segmentation and selection techniques (*Raycast* and *CutPlane*). They evaluate their effects on user efficiency in function of the difficulty of the selection task. The *AGENT model* was made and the code was generated. Section 5.2.1 presents the models we made from this work (Figures 10 and 11). Section 5.2.2 presents the code integration process for producing the complete experiment.

5.2.1 Modeling

Figure 10 shows the *experimental conditions model* made with AGENT, from the work of Mossel *et al.*. In the presentation of their experiment, Mossel *et al.* give clearly the independent variables *"The participants had to use both* Raycast *and [...]* CutPlane *[...] in combination with all three*

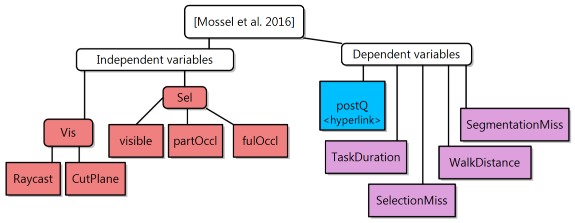


Figure 10: *experimental conditions model* from the experiment of Mossel *et al.* [22].

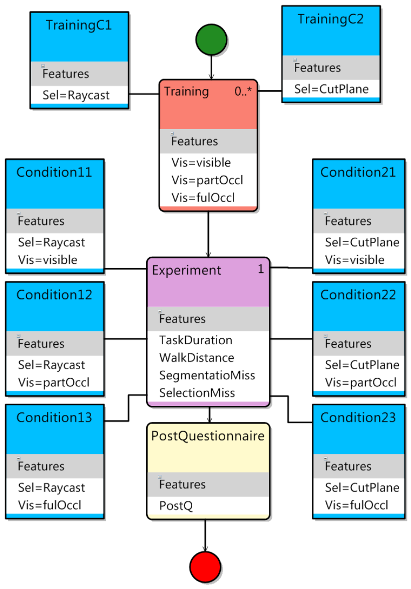


Figure 11: *protocol model* from the experiment of Mossel *et al.* [22].

scenarios". The three scenarios each correspond to a level of difficulty: selecting a *Fully visible*, *Partially occluded*, or *Fully occluded* object. Hence, there are two enumeration variables (*Vis* and *Sel* on Figure 10) with respectively two and three values: *Raycast, CutPlane*; and *visible, partOccl, fulOccl*. The dependent variables are then explicitly given by Mossel *et al.*. The *"Objective Performance Measures"* are: *TaskDuration, WalkDistance, SegmentationMiss, SelectionMiss*. The *"Subjective Performance Measures"* are gathered in a post-questionnaire we identified by *postQ* in Figure 10.

Figure 11 shows the *protocol model*. The phases are: *"training phase, [...] experiment, and [...] a post-questionnaire"*. The three phases are represented with the three states of the *protocol model* in Figure 11: *Training, Experiment, PostQuestionnaire*. *PostQuestionnaire* is a simple state because the participant only has to answer to the questions. The *Experiment* state is a random-loop of multiplicity 1 with six conditions combining the values of the two independent variables (see Figure 11). The justification can be found in two sentences from Mossel *et al.*: *"The participants had to use both* Raycast *and* CutPlane *in combination with all three scenarios. That results in total in six different tasks which the participants perform in random order"*. *Training* corresponds to an acclimatization phase. Mossel *et al.* give this description: *"[Participants] were freely interacting in a test environment, which comprised a simple Unity3D scene with some artificial virtual objects that could be segmented and selected and where objects' visibility ranged from visible to fully occluded. As soon as the user reported to feel confident, the experiment stage [...] started"*. Hence, *Training* is a customized-loop state with unlimited multiplicity ($0..*$). All visibility conditions are presents in the Virtual Environment (*visible, partOccl, fulOccl*). The user can switch between the two selection techniques, which explains the two conditions *TrainingC1* and *TrainingC2* in Figure 11.

5.2.2 Code Integration

With the *feature binding library*, the experiment designers can bind the independent variables to the Unity components that manage them. In the state *Training*, the three visibility conditions are selected at the same time. It just means that they coexist at the same time in the test environment. The objective independent variables (*TaskDuration, WalkDistance, SegmentationMiss, SelectionMiss*) are to be bound to numeric calculated fields (*i.e.,* fields in Unity scripts) that are updated automatically by the VR application and that correspond to the task duration, the walk distance, the number of inappropriate segmentation attempts, and the number of inappropriate selection attempts, for each trial.

With the *condition sequencing library*, the experiments designers can chose the implementation they want for the random-loop state. They can also make the *Training* loop state end by detecting the user request, performed on the UI. Switching between the conditions *TrainingC1* and *TrainingC2* is also made by detecting a user request.

With the *trial completion management library*, the experiment designers can precise the trial end condition: the user performed the segmentation and selection task. It can be for example detected through a Unity script. The action to perform to begin the new trial is to reset the position of the user to the start position.

5.3 Discussion

AGENT is made to work with Unity. However it is possible to adapt it to other VR platforms (*e.g.,* CRYENGINE, Unreal, *etc.*). The modeling part of AGENT does not need to be modified. We estimate that the *compiler* needs to be partially re-implemented. The *integration module* needs to be totally re-implemented. Nevertheless we estimate that the induced effort is minimal. These re-developed components are indeed reusable and they remain small.

The *compiler* needs only to translate the protocol model to a state machine. Estimating the effort for implementing The *integration module* is more difficult because it highly depends on the target platform. However, if we base our estimation on our implementation, the only task is to develop a dozen of classes, each one containing no more than ten lines of fields and functions definitions.

Furthermore, producing an experiment with our implementation can be done in few hours. To integrate the code, some functions must be implemented (see Section 5.1.3). We estimate that five lines of code for each of them in average is a maximum.

6 Conclusion and Future Works

In this paper, we presented an approach for the automatic generation of experimental protocol runtime. We conducted a study on fifteen experiments reported in the VRST'16 proceedings [1], to determine the properties our approach should satisfy. Seven properties could be drawn up. These properties define the main concepts our approach has to take into consideration: independent and dependent variables; between and within subject factors; acclimatization, calibration, and data acquisition phases of the protocol. They also highlight the diversity of the independent and dependent variables in experiments: variables can be software or hardware features, measurements or subjective analysis, data can be produced by very specific devices, *e.g.,* for measuring physiological constants.

We designed an approach conform to these properties. We then introduced the AGENT DSL, that generates the experimental protocol runtimes. More particularly, AGENT allows to write *experimental conditions models* and *protocol models*. These models are then compiled into *runnable code*, for integration in a VR project.

We demonstrated that our approach is conform the seven properties deduced from the preliminary analysis. The *experimental conditions model* indeed allows to define independent and dependent variables, and the *protocol model* allows to define acclimatization, standardization, and data acquisition phases. Between and within subject factors are implicitly defined by the use of several *protocol models*. Finally, the *integration module* allows to bind independent and dependent variables to any hardware or software resource, hence allowing to handle their potential diversity.

We demonstrated the feasibility of our approach by using AGENT on two reported experiments. One of them was completely rebuilt and for the other one we generated the code. The 15 experiments reported in the VRST'16 proceedings [1] can be generated with the usage of AGENT.

In future works, we have the intention to submit AGENT to a user-study to evaluate its efficiency over manual implementation. We also plan to evaluate its replicability with engines different from Unity. Moreover, we plan to extend this work to the statistical analysis step, by performing it automatically after the experiment was conducted. Finally, we plan to investigate further the automatic production of VR applications. In other domains than experiments, the production of VR applications is indeed done manually, without code or concept reuse. In particular, we plan to propose an approach for automatically generating VR applications for training.

References

[1] *VRST '16: Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, New York, NY, USA, 2016. ACM.

[2] D. A. Bowman, D. B. Johnson, and L. F. Hodges. Testbed Evaluation of Virtual Environment Interaction Techniques. In *VRST '99: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 26–33. ACM, 1999.

[3] J. Brooke et al. Sus-a quick and dirty usability scale. *Usability evaluation in industry*, 189(194):4–7, 1996.

[4] A. M. Colman. *A dictionary of psychology*. Oxford University Press, USA, 2015.

[5] J. W. Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, 2013.

[6] F. T. Durso, A. R. Dattel, S. Banbury, and S. Tremblay. Spam: The real-time assessment of sa. *A cognitive approach to situation awareness: Theory and application*, 1:137–154, 2004.

[7] A. Field. *Discovering statistics using SPSS*. Sage publications, 2009.

[8] A. Field and G. Hole. *How to design and report experiments*. Sage, 2002.

[9] M. Fowler. *Domain-specific languages*. Pearson Education, 2010.

[10] J. L. Gabbard, D. Hix, and J. E. Swan. User-centered design and evaluation of virtual environments. *IEEE computer Graphics and Applications*, 19(6):51–59, 1999.

[11] J. Grübel, R. Weibel, C. Hölscher, and V. R. Schinazi. EVE: A Framework for Experiments in Virtual Environments. In *Proceedings of Spatial Coginition 2016*, 2016.

[12] S. G. Hart and L. E. Staveland. Development of nasa-tlx (task load index): Results of empirical and theoretical research. *Advances in Psychology*, 52:139 – 183, 1988. Human Mental Workload.

[13] K. Hornbæk. Current practice in measuring usability: Challenges to usability studies and research. *International Journal of Human-Computer Studies*, 64(2):79 – 102, 2006.

[14] W. Huang, L. Alem, and M. A. Livingston. *Human factors in augmented reality environments*. Springer Science & Business Media, 2012.

[15] V. Interrante, B. Ries, and L. Anderson. Distance Perception in Immersive Virtual Environments, Revisited. In *IEEE Virtual Reality*, pages 3–10. IEEE, 2006.

[16] S. Julier, S. Feiner, and L. Rosenblum. *Mobile augmented reality: a complex human-centered system*. Springer Science & Business Media, 2001.

[17] M. E. Latoschik, J.-L. Lugrin, and D. Roth. Fakemi: A fake mirror system for avatar embodiment studies. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 73–76, New York, NY, USA, 2016. ACM.

[18] G. Le Moulec, F. Argelaguet, A. Lécuyer, and V. Gouranton. Take-over control paradigms in collaborative virtual environments for training. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 65–68, New York, NY, USA, 2016. ACM.

[19] D. Medeiros, M. Sousa, D. Mendes, A. Raposo, and J. Jorge. Perceiving depth: Optical versus video see-through. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 237–240, New York, NY, USA, 2016. ACM.

[20] A. Meier, H. Spada, and N. Rummel. A rating scheme for assessing the quality of computer-supported collaboration processes. *International Journal of Computer-Supported Collaborative Learning*, 2(1):63–86, Mar 2007.

[21] M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dec. 2005.

[22] A. Mossel and C. Koessler. Large scale cut plane: An occlusion management technique for immersive dense 3d reconstructions. In *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*, VRST '16, pages 201–210, New York, NY, USA, 2016. ACM.

[23] J. Pearl, E. Bareinboim, et al. External validity: From do-calculus to transportability across populations. *Statistical Science*, 29(4):579–595, 2014.

[24] D. C. Schmidt. Model-driven engineering. *COMPUTER-IEEE COMPUTER SOCIETY*, 39(2):25, 2006.

[25] T. Schubert, F. Friedmann, and H. Regenbrecht. The experience of presence: Factor analytic insights. *Presence: Teleoperators and Virtual Environments*, 10(3):266–281, 2001.

[26] W. R. Shadish, T. D. Cook, and D. T. Campbell. *Experimental and quasi-experimental designs for generalized causal inference*. Wadsworth Cengage learning, 2002.

[27] M. Slater. Place illusion and plausibility can lead to realistic behaviour in immersive virtual environments. *Philosophical transactions of the Royal Society of London. Series B, Biological sciences*, 364(1535):3549–3557, 2009.

[28] M. Slater, M. Usoh, and A. Steed. Depth of presence in virtual environments. *Presence: Teleoperators and Virtual Environments*, 3(2):130–144, 1994.

[29] K. M. Stanney, M. Mollaghasemi, L. Reeves, R. Breaux, and D. A. Graeber. Usability engineering of virtual environments (ves): identifying multiple criteria that drive effective ve system design. *International Journal of Human-Computer Studies*, 58(4):447 – 481, 2003.

[30] J. G. Tromp, A. Steed, and J. R. Wilson. Systematic usability evaluation and design issues for collaborative virtual environments. *Presence: Teleoperators and Virtual Environments*, 12(3):241–267, 2003.

[31] A. van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *SIGPLAN Not.*, 35(6):26–36, June 2000.

[32] D. Wile. Lessons learned from real dsl experiments. *Science of Computer Programming*, 51(3):265 – 290, 2004.

[33] B. G. Witmer and M. J. Singer. Measuring presence in virtual environments: A presence questionnaire. *Presence: Teleoperators and Virtual Environments*, 7(3):225–240, 1998.