Vrije Universiteit Brussel

FACULTY OF SCIENCE
Department of Computer Science
Web & Information Systems Engineering

Towards a Graphical Scenario Modeling Language for Virtual Environments

Thesis submitted in fulfillment of the requirements for the degree of Licentiate in Applied Computer Science

Jeroen Wauters

Academic Year 2006-2007

Promotor : Prof. Dr. Olga De Troyer
Supervisor : Bram Pellens

Vrije Universiteit Brussel

FACULTEIT VAN DE WETENSCHAPPEN
Departement Computerwetenschappen
Web & Information Systems Engineering

Naar een Grafische Taal voor het Modelleren van Scenario's voor Virtuele Omgevingen

Proefschrift ingediend met het oog op het behalen van de graad van Licentiaat in de Toegepaste Informatica

Jeroen Wauters

Academiejaar 2006-2007

Promotor : Prof. Dr. Olga De Troyer
Begeleider : Bram Pellens

Abstract

With the increase in power and graphical capacities of modern day computers, 3D virtual reality applications are becoming available to the general public, giving rise to an increased need for fast design and implementation of virtual environments. Hence, VR-DeMo was conceived, a methodology for the conceptual modeling of virtual environments. In its current state, VR-DeMo provides support for modeling the static scene of a virtual world, the objects populating it, the behaviors performed by these objects, and user interaction with the world.

This thesis discusses VR-SDL, an extension to the existing VR-DeMo methodology, for modeling interactive scenarios for a virtual world. Scenarios are a missing link in the current VR-DeMo approach : rather than modeling the virtual environment itself, they model the purpose of the virtual environment, and (parts of) the application for which it is used.

In VR-SDL, actors participating in the scenario are modeled separately from the actual scenario, using a technique called actor graphs – a special type of finite state machine. Consequently, the scenario itself is modeled using a scenario graph, a special kind of flowchart, describing the possible interaction between actors during the scenario, and the results thereof.

Abstract

Door de toenemende processorkracht en grafische capaciteiten van hedendaagse computers worden applicaties van virtuele realiteit stilaan beschikbaar voor het grote publiek. Hierdoor stijgt de nood aan een snelle manier om virtuele omgevingen te modelleren en implementeren. Om deze reden werd VR-DeMo gecreëerd, een benadering voor het conceptueel modelleren van virtuele omgevingen. In zijn huidige toestand biedt VR-DeMo ondersteuning voor het ontwerp van de statische opbouw van een virtuele wereld, de objecten die erin voorkomen en hun gedrag, en interactie van de gebruiker met de wereld.

In deze thesis wordt VR-SDL besproken, een uitbreiding op de bestaande VR-DeMo benadering, voor het modelleren van interactieve scenario's voor een virtuele wereld. Scenario's zijn een ontbrekende schakel in de huidige VR-DeMo benadering : zij dienen niet voor het modelleren van de virtuele omgeving zelf, maar voor het doel ervan, en (delen van) de applicatie waarvoor de virtuele wereld wordt gebruikt.

In VR-SDL wordt een onderscheid gemaakt tussen het ontwerp van de actoren die deelnemen aan een scenario, en dat van het scenario zelf. De actoren worden gemodelleerd door middel van actor graphs, een speciaal soort finite state machine. Vervolgens wordt het scenario gemodelleerd door middel van een scenario graph, een speciaal type flowchart, dat de mogelijke interactie tussen actoren beschrijft, evenals de gevolgen hiervan.

Acknowledgements

First and foremost, I would like to thank **Bram Pellens** from the WISE lab, for his guidance during my final year – not only while I was working on my thesis, but on my apprenticeship as well. He provided me with numerous interesting papers, discussed my research results with me for hours on end, read my temporary proposals before each meeting (as incoherent and last-minute as they may have been), reviewed and commented on my final write-up, and much more. Without his assistance, the quality of this thesis would undoubtedly have been significantly lower.

Next, there are two other WISE members who deserve my gratitude. Firstly, Dr. **Frederic Kleinermann**, for following up on my work, providing many helpful hints and suggestions, reviewing my thesis, and lending me a few manuscripts from his apparently vast VR library. Secondly, my promotor, Prof. Dr. **Olga De Troyer**, for giving me the opportunity to write a thesis in the field that interests me the most.

I am also grateful to **my parents**, for granting me the freedom to pursue my path of choice, funding my education, and supporting me along the way. The latter also goes for **my friends**, with a special mention for the good people of the **Electric Hellfire Club** : thanks for allowing me to bail on you guys for months when I needed to, and welcoming me back afterwards.

And finally, to conclude in a somewhat unorthodox fashion, I would like to thank the Belgian singer and comedian **Urbanus**. His songs were pretty much the only thing that kept me going at the end of my third year. If it hadn't been for my vast Urbanus-playlist, I doubt I would have ever made it past that infernal exam period.

Contents

List of Figures

Chapter 0

Introduction

0.1. Virtual Reality

Upon hearing the term ‘virtual reality’ (VR), the layman might immediately think of science fiction scenes from books or movies, where people are entirely immersed in life-like virtual worlds through exotic equipment such as visors, sensor suits, neural plugs, and so on. This Hollywood-inspired misconception conceals the fact that virtual reality is much more than a mere futuristic vision, it is a very real research field, with a variety of contemporary applications. In general, VR is the technology for creating and interacting with real-time, three-dimensional, computer simulated environments.

This definition will probably sound significantly more familiar : three-dimensional, real-time environments have been commonplace in regular households for over ten years, due to gaming computers and consoles. In 1994-1995, with the rise of the fifth generation of videogame consoles, most notably represented by the *Sony PlayStation*, *Nintendo 64* and *Sega Saturn*, 3D virtual environments began surfacing in people’s living rooms. To date, videogames are still the most common way for people to experience virtual reality firsthand.

The range of applications of VR extends far beyond games, however. Computer simulations are being used in various training programs, while avoiding the downsides a real training would entail, such as a high risk of physical harm, a prohibitively high cost, or simply an inability to recreate the training in reality due to laws of physics. Examples include army ground troops training for urban combat using combat games [30], pilots training in flight simulators [28], or aspiring surgeons performing virtual surgery [14], without the risk of accidentally killing a patient due to a rookie mistake. While these simulations can not (and are not intended to) eliminate the need for real training, they do offer a very valuable addition to the training process.

Evidently, VR simulations can also be used for different purposes, like crash-testing a car without actually having to crash it. Additionally, VR is known to be used for therapeutic purposes, particularly for phobia treatment, by confronting subjects with their fears in a controlled environment.

Another use is publicity : shop owners can offer a 3D model of their store and merchandise on their website, providing the surfer with a feel for the look and atmosphere, and more importantly, visualization of the products for sale – more so than a picture could achieve. Real estate agents can use the same technique to offer a prospective client an initial view of their dream home, without them actually having to visit it.

With the continuous increase in power and graphical capacities of modern day computers, VR applications are becoming increasingly more accessible to the general public. Hence, new types of applications, even more general than before, are surfacing : virtual tour guides, leading the spectator on a tour through a 3D replica of a building or city, advertisements as a promotion for a certain brand, incorporated in a popular virtual environment (like an MMOG[1]), and so on. An example of the latter is the popular science-fiction MMOG *Anarchy Online* (AO) [11], where non-paying customers see in-game advertisements for real products (*figure 1*). The recently announced *Home* [27] application for *Sony*'s *PlayStation 3* console is another nice example of the broadening use of VR : rather than a game, it is a virtual meeting place, where users create their own home, watch movies, and talk to their friends, all while controlling a 3D avatar of themselves.

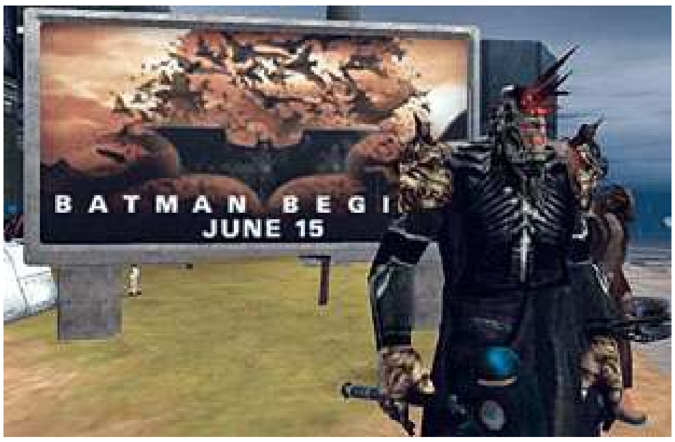


Figure 1 – An in-game billboard advertising the movie *Batman Begins* in AO

[1] Massively Multiplayer Online Game, a game played by thousands of interacting players at the same time.

0.2. VR-DeMo : Modeling Virtual Reality

With the growing importance of virtual reality, the need for an easy, accessible technique to model virtual environments (VEs) seems obvious. One way to achieve this is the introduction of a conceptual modeling approach : such a technique would allow domain experts in the subject of the virtual environment to perform the modeling, and in a later stage, even allow for generation of the entire VE based on these models. This is both time- and cost-effective, due to the elimination of the communication overhead between the designer (the domain expert) and the implementer (a programmer) of the VE, as these two roles can now be fulfilled by the same person. Unfortunately, the focus of past VR research has been on technological advancement, rather than modeling of VEs. While this is certainly understandable – it makes no sense to be able to draw intricate plans for a house which could not be built anyway, due to a lack of feasible resources – it also implies that a standard VR modeling approach does not yet exist.

Therefore, the Flemish subsidy organization (IWT) funded a project where two Flemish universities, the *Vrije Universiteit Brussel* (VUB) and *Universiteit Hasselt* (Uhasselt), cooperate in trying to create such a (graphical) modeling technique. The resulting technique is dubbed ***VR-DeMo*** (Virtual Reality : Conceptual Descriptions and Models for the Realization of Virtual Environments). The long-term goal of VR-DeMo is to provide an intuitive way to model virtual environments, and automatically generate the code for the VE, based on these models. *Figure 2* shows an overview of the approach taken to model a VR application using VR-DeMo.

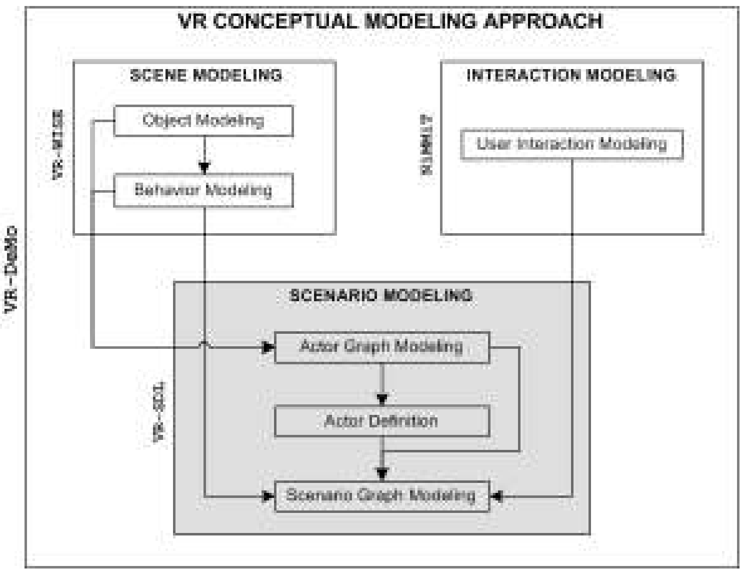


Figure 2 – Conceptual modeling approach for designing a VR application with VR-DeMo.

The rectangles represent the different modeling techniques present in VR-DeMo. If a rectangle *A* is connected to a rectangle *B*, this means that the modeling technique *B* is dependent on the output of technique *A*. The grey box represents the additions suggested in this thesis.

The VUB-part of the research, conducted by the research group *Web & Information Systems Engineering* (WISE), is called ***VR-WISE*** [36]. VR-WISE is focused on modeling the static scene of the virtual environment, the objects therein, and behaviors attributed to these objects. The part of the research conducted in Hasselt primarily deals with user interaction, for which an approach called ***NiMMiT*** (Notation for Multimodal Interaction Techniques) [29] was introduced.

This thesis provides an extension to the research described, by attempting to add a graphical approach for modeling interactive scenarios in a virtual environment. Throughout the thesis, this approach will be referred to as ***VR-SDL*** (Virtual Reality Scenario Description Language).

0.3. VR-SDL : Scenarios in Virtual Reality

0.3.1 Motivation

In its current form, VR-DeMo provides support for modeling a dynamic virtual environment, complete with objects executing complex behaviors (through VR-WISE), and for modeling user interaction with VEs, via menus and interfaces (through NiMMiT). Hence, the necessary means to model an interactive VE are already present.

However, this is clearly insufficient. A virtual environment where nothing happens aside from a few pre-programmed behaviors is not very interesting : all VEs are designed with a goal in mind, they are part of some application, whether it is a game, a training simulation, an online shop, or something else. Therefore, there is an obvious need for being able to model application-relevant occurrences in the VE. For example, in a game, the designer should be able to specify what actions the player should take to finish a level, and how his actions influence the VE. In a surgical training simulation, one should be able to model the possible mistakes that can be made, and their influence on the outcome of the operation. In an online shop, the way in which the user can interact with the products (rotate, recolor, retexture, ...) should be defined.

It is precisely this missing link which is introduced by VR-SDL, the scenario description language described in this thesis. VR-SDL is much more closely tied to VR-WISE than to NiMMiT, since it describes manipulations of the objects modeled by VR-WISE.

0.3.2. What is a Scenario?

Before introducing any specifics about the VR-SDL approach, it is imperative to understand the scope of the problem that will be tackled. The general idea was described in the previous section, but it is not so obvious to grasp exactly what constitutes a scenario.

Indeed, scenarios can be viewed from many different perspectives. When searching the literature on the subject, it quickly became apparent that different authors have different views on what a scenario is. Richard Wages [13, 31-33], for example, looks at scenarios from a storytelling perspective. In his work, 'scenario' is synonymous to 'story', and they are modeled using a branching story graph. Peter Jason Willemsen, who has developed a scenario modeling language in his PhD thesis [35], takes an entirely different approach to scenarios. In Willemsen's work, a scenario is observed from a behavioral point of view. For example, a scenario could be a car driving onto an intersection at the exact moment the player enters it from the opposite direction, and the ensuing events. However, neither Wages nor Willemsen explicitly define the notion of a scenario.

In the context of VR-DeMo, an approach directed towards automatic generation of source code based on conceptual models, Willemsen's view is largely preferable. Wages' take on scenarios is too informal, his work is more akin to creating a tool for writing nonlinear stories, than creating a scenario modeling language. Therefore, in VR-SDL, scenarios are defined more formally, as a web of possible sequential events.

The above leaves us to question the distinction between a behavior and a scenario. Intuitively, a scenario is "something that happens in the virtual world" – but so is a behavior. The obvious reply to this would be that behaviors are executed by only one object, while scenarios involve many objects. Unfortunately, this is not necessarily the case. Behaviors in VR-DeMo can be defined as interactions between multiple actors. For instance, one could define a chase/evade behavior with two actors, A and B, where A tries to capture B, while B attempts to evade A. While A and B clearly perform different actions, the chase/evade sequence is perceived, modeled and implemented as a single behavior. On the other hand, while it is true that scenarios will usually contain many interacting objects, this is by no means a requirement.

Another defining factor might be user interaction. After all, the user's actions usually influence the course and outcome of a scenario. Although this is true in general, it is also possible to define a scenario the user is not involved in, relying on the AI of the actors to determine the next action to be taken. Additionally, behaviors need not be void of user interaction. In the chase/evade example above, either of the actors A or B could in fact be controlled by the user, prompting all of his actions to influence the execution of the behavior.

Finally, there is the scene. While behaviors can trigger changes in the actors performing them (such as sound effects or animations being played), they have no way to influence the static part of the scene. Scenarios do have this option at their disposal. However, since in VR-DeMo, all parts of the static scene are modeled as objects on the lower level (as will be explained in chapter 2), they could all be used as actors in the definition of a behavior.

In conclusion, the distinction between behaviors and scenarios is not an obvious one. A situation where an actor A is patrolling some route until the player talks to it, at which point it will start to flee from the player, can both be viewed as a behavior and as a scenario. So, rather than attempting to formulate a formal definition for a VR scenario, some general guidelines are provided :

- The more **user interaction** influences the execution, the better it is to model a scenario instead of a behavior.
- A **large effect on the scene** generally means it is preferable to model the situation as a scenario, rather than a behavior.
- A **very complex situation** is usually simpler to model as a scenario, than as a behavior.
- Many actors, and particularly **many different types of actors**, means a scenario is the way to go.
- When there is little interaction or influence on the scene, but only a few different types of actors, performing a lot of low-level actions, a behavior is preferred.

These guidelines imply that when designing the scenario modeling technique, particular attention should be paid to the ability to represent user and scene interaction. Take note, however, that the scope of scenarios is somewhat wider than simply behavior and interaction. In a sense, they are the encasing technique to define a purpose for all the components in a VE, from the perspective of some application.

0.3.3. Approach

From the discussion above, the basic components of a scenario can be derived : scenarios consist of a number of objects interacting with one another (the *actors* of the scenario), while possibly influencing the environment in the process. Therefore, the proposed notation uses two different types of graphs.

Firstly, ***actor graphs*** model the objects participating in a scenario. Rather than describing their static structure – a property for which a modeling technique already exists in VR-DeMo – actor graphs describe the states of the object, relevant to the scenario. Within different states, objects can display different behaviors, visuals or properties. Hence, actor graphs are modeled using a special type of finite state machines.

Modeling all participants of a scenario obviously does not equal modeling a scenario. An additional graph is required for this, describing the actions by and interactions between different objects, and their effect on the environment – in other words, on the other objects. This graph is called a ***scenario graph*** : essentially, it is a flowchart, depicting the possible courses of the scenario.

0.4. Scope and Structure of this Thesis

0.4.1. Scope

The reason why this thesis is rather lengthy, is because it is intended as a comprehensive overview for the reader who wants to grasp the VR-SDL notation in its entirety. Every available construct is explained in detail, and reasons for the necessity of more exotic constructs are given. Moreover, many examples are provided, making it easier to understand the practical use of all these constructs. Additionally, two chapters about the background of scenario modeling in general, and VR-DeMo in particular, are included for the interested reader.

Due to the complexity of the full VR-SDL modeling language, there is a danger that the reader might lose track of the general picture. To avoid this, the *VR conceptual modeling* diagram from *figure 2* is used as a recurring theme throughout the thesis. Each relevant chapter opens with this diagram, where the elements described in that chapter are marked. Parts of the diagram, expanded to show more detail about a certain approach, also return in several relevant sections.

Naturally, not all readers will be interested in an all-embracing description of the notation. To provide support for people who are simply looking to get a general overview of the VR-SDL approach, without going into a lot of detail, the introduction to each chapter contains some reader guidelines. These guidelines denote the most important sections of the chapter, and explain which sections may be skipped without losing track of the notation in general.

0.4.2. Structure

Globally, this thesis consists of two major parts : *Background*, and *Modeling Scenarios in VR-DeMo*.

0.4.2.1. Background

This part contains all information the reader needs to be able to understand the second part. It is itself split up into two chapters : the first one, *State of the Art*, describes the current state of research in modeling scenarios for VR. Some literature on the subject is described, including a number of other attempts to formalize a scenario description language. These approaches are compared to the one described in this report, and arguments are provided why VR-SDL is a useful contribution to the field of scenario modeling. Additionally, an attempt at formalizing the notion of a scenario is made. The second chapter, *VR-DeMo: An Overview*, briefly summarizes the part of the VR-DeMo approach relevant to understanding the second part of the thesis.

0.4.2.2. Modeling Scenarios in VR-DeMo

The results of the research performed, and new contributions to the VR-DeMo modeling approach, are described in this part. It consists of four chapters : initially, the *Methodology* chapter describes the VR-SDL approach in general, to provide the reader with an overview of how the technique works. The main chunk of this part concerns the chapters *Actor Graphs* and *Scenario Graphs*, providing a detailed description of these two components of VR-SDL. Finally, the chapter *Conclusion and Future Work* provides a summary of the approach, along with some pointers for future extensions.

Part I

Background

Chapter 1

State of the Art

1.1. Introduction

In this chapter, an overview of the literature consulted in preparation for this thesis is provided. Initially, some light is shed on the state of the art in scenario modeling. Afterwards, a number of similar modeling approaches are considered, and compared to the work performed, as described in part 2 of the thesis. Furthermore, an attempt is made to draw some conclusions from the literature, concerning some points of focus for designing a scenario modeling language, and pitfalls that should be avoided.

Since no new work is introduced in this chapter, it may be skipped by readers who are simply looking to get an insight into the VR-SDL approach. It is mostly interesting for those interested in the field of scenario modeling in general, rather than scenario modeling in VR-DeMo.

1.2. Legacy of VR Research

Research in Virtual Reality has been conducted ever since 3D rendering was possible, but even today, most of it is aimed at the technology, rather than a design methodology. From the VR-WISE website [36] :

> *"Indeed, most of the research today in the area of virtual reality happens in its deployment technology: hardware (dedicated computers, head mounted displays, gloves, etc.), visualisation algorithms and techniques; languages for interactive 3D applications (VRML, X3D, Java 3D, etc.); tools for building 3D environments, browser extensions and plugins for 3D; and several protocols like VRTP (Virtual Reality Transport Protocol), behaviour interaction protocols like DIS (Distributed Interactive Simulation). The virtual environment itself is usually designed in an ad hoc way; in fact little or no guidelines or methods are available for this."*

Of course, considering that VR is still relatively new[2], this is quite understandable : one would not develop a revolutionary way to draw plans for a house without first making sure the bricks are solid and the mortar is sticky. And, as with any technology, improvements will always be possible, and thus, research is always relevant.

However, we have arrived at a stage where the groundwork is solid enough to build professional applications that cost a small fortune to create, but have the potential to generate a massive profit. Consider the example of the commercial game industry [34] : its yearly income exceeds that of Hollywood, while the costs of creating a single game have skyrocketed, thus creating an extremely high-risk – and potentially high-revenue – environment. Game producers do what they can to minimize risks and the probability of flops, and in that context, it seems bizarre that most game developers do not seem to use a formal methodology to design their games. There is a gap between the game's designers, who brainstorm about the game's atmosphere and gameplay, and its programmers and level creators, who use their own artistic skills to create the content proposed by the designers. The whole process is comparable to building a cathedral by telling the foreman what it should look like, and allowing him to follow his instincts from there on out.

It is clear that there is a missing piece in this puzzle, namely the VR design methodology. Analogous to UML for software programs, there should be a similar modeling technique for virtual reality applications.

1.3. Finite State Machines

As section *1.4* will show, finite state machines (FSMs) are a popular approach to modeling scenarios. FSMs are diagrams consisting of two main building blocks : *states*, and *transitions* between these states. They have a very wide range of applications, going from AI programming to building circuits in electrical engineering.

FSMs can contain actions, which form the basis of a theoretical distinction between *Mealy* and *Moore* machines. In a Mealy machine, actions can be performed inside the states, while in a Moore machine, actions are performed on the transitions. Although Mealy machines usually reduce the number of necessary states, in practice a mixture form between both forms is often used, which eases the

[2] When compared to certain other fields in computer science, like programming language design, anyway.

modeling, and leads to more straightforward diagrams. UML state machine diagrams [10] are an example of a mixture form.

FSMs display some interesting properties with regards to scenario modeling. They are formal, (usually) intuitive to interpret, and appear particularly useful to model the evolution of objects participating in a scenario. A popular example for demonstrating the use of FSMs is a door lock : put the key in and turn it and the door becomes unlocked, turn it the other way around and it becomes locked. This could be seen as a (partial) scenario, or at least, as the behavior of a door in some scenario.

However, pure FSMs are too unwieldy to use for modeling large scenarios : they provide little support for handling data, they have trouble modeling many different objects changing states together, they can not be in more than one state at the same time, which leads to excessively large diagrams when many objects are present, and so on. Additionally, a scenario language for use within VR-DeMo should be integrated in that approach, and provide support for handling the constructs presented by the different branches of VR-DeMo.

Therefore, an entirely new scenario modeling language will be created, called VR-SDL. The popularity of FSMs in scenario modeling approaches, and their useful properties will be taken into account when designing VR-SDL, however.

1.4. Research in Scenario Modeling

Evidently, some research has been performed on the subject of VR modeling in general, and scenario modeling in VR in particular. In this section, some scenario modeling approaches are described, and their relevance to and differences from VR-SDL are discussed.

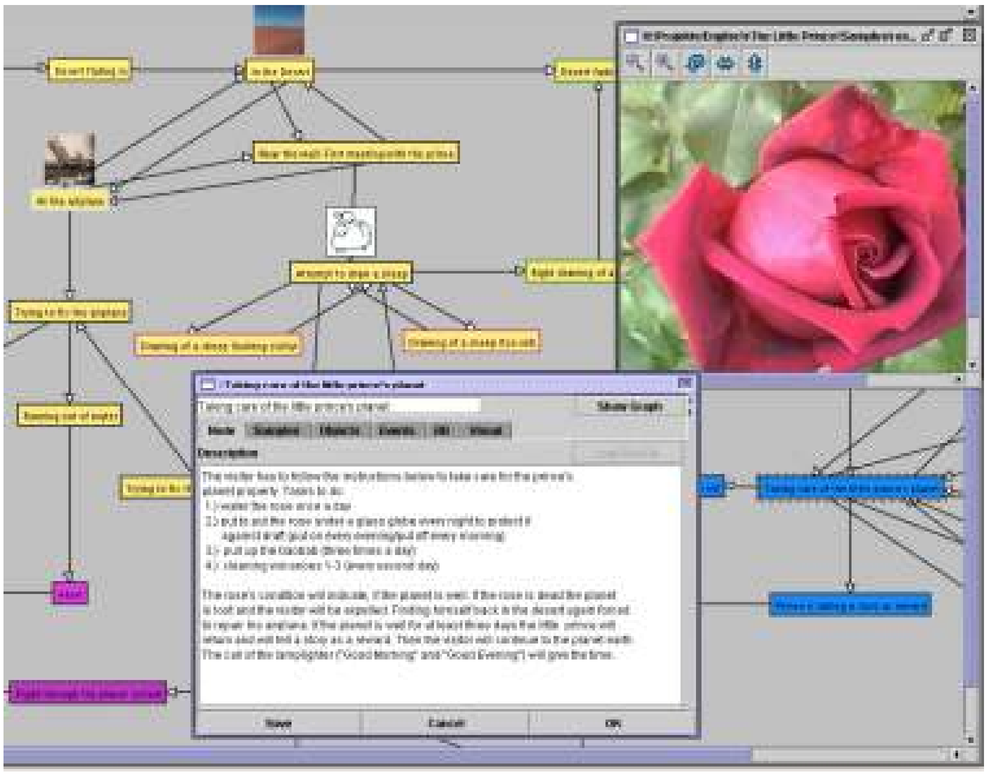
1.4.1. alVRed

1.4.1.1. Description

alVRed [13, 31-33] is a project by Wages (mentioned earlier in section *0.3.2*), Grützmacher and others, aimed towards the creation of a set of tools for designing non-linear, interactive stories in virtual environments.

As discussed before, the alVRed approach considers ‘scenario’ to be synonymous to ‘story’, and thus, the focus of the work lies in creating an authoring tool for scriptwriters. The tool allows for authors to model objects participating in the story, by creating a number of connected, story-relevant states for the objects. The story itself is seen as an object as well, and is modeled in the same way.

The object states are little more than (possibly nested) nodes wherein an author can type – in prose – a description of the state. Additionally, media such as pictures, video or audio can be linked. State transitions can be scripted, using a very simple scripting language. Hence, the nodes basically serve as a structured storyboard for the designers, who use the actor graphs to implement the story structure. An example of a (partial) actor graph is given in *figure 3*.



Taking care of the little prince's planet

The visitor has to follow the instructions below to take care for the prince's planet properly. Tasks to do:
1.) water the rose once a day
2.) put to and the rose under a glass globe every night to protect it against draft (put on every evening/pull off every morning)
3.) pull up the baobab (three times a day)
4.) cleaning volcanoes 1-3 (every second day)

The rose's condition will indicate, if the planet is well. If the rose is dead the planet is lost and the visitor will be expelled. Finding himself back in the desert again forced to repair the airplane. If the planet is well for at least three days the little prince will return and will tell a story as a reward. Then the visitor will continue to the planet earth. The call of the lamplighter ("Good Morning" and "Good Evening") will give the time.

Figure 3 – A screenshot from an actor graph in the alVRed authoring tool

1.4.1.2. Relevance

The standard approach of writing down a story, storyboard or screenplay might work for linear scenarios, such as those used in television or film, but it is insufficient for the dynamic, non-linear stories proper to VR. Therefore, the work performed in the alVRed project is centered largely around providing scenario writers with a structured way to specify non-linear stories.

While alVRed seems to be more informal than the work that will be attempted for this thesis, some of the ideas discussed are very interesting from the perspective of a graphical scenario modeling language. The fact that scenarios contain a variety of different actors, who should themselves be modeled, is an interesting perspective, and one that is used in the extension to VR-DeMo as well. Additionally, the way in which the object modeling is specified – by using extended finite state machines – was adopted in VR-SDL as well.

The fact that alVRed considers the scenario to be merely a special type of object, is a notable difference to the approach taken in this thesis. Early tests with this type of notation led to the conclusion that, while it might be sufficient for an informal scenario description, a scenario model that should be usable for code generation requires a more detailed specification.

1.4.2. EDF + SDL

1.4.2.1. Description

Environment Description Framework (EDF) and Scenario Description Language (SDL) are two parts of a joint modeling technique for virtual urban traffic situations, presented in the PhD thesis of Peter Jason Willemsen [35].

EDF is the part of the modeling technique concerned with modeling the static portion of the scene : it describes how to model roads, sidewalks, intersections, and related objects. The modeling is based on certain building blocks, such as road segments, that are adjustable through the use of parameters (e.g. the length or curvature of a road segment) and connectable through connection points (called junctures). The technique is very intuitive, and reminiscent of construction toys, like Lego.

SDL is a scripting language to describe scenarios in the world. Its syntax is very similar to Java or C++, but evidently contains only a limited amount of available constructs and operators. Two types of statements are distinguished : ***activities*** and ***monitors***. Activities are simple commands, such as 'change lane', or 'speed up'. Monitors are statements controlling the activities' temporal conditions, in other words, they determine when the activities occur. Four commands are provided, varying in two dimensions : the amount of execution times (once or periodic), and the length of one execution (impulsive or continuous). Additionally, statements for creating or destroying objects in real time are available. Below is a short code snippet, demonstrating the SDL syntax :

```
aslongas( vehicle.road_type() == FOUR_LANE_ROAD )
   {
      whenever( vehicle.speed() < threshold )
         {
```

```
                    send vehicle increase_speed( 0.10 )
                }
            }
```

The interpretation is straightforward : as long as the vehicle is driving on a four lane road, its speed is increased by 0.10 whenever it goes below a certain threshold. Note that the code displays this almost literally.

1.4.2.2. Relevance

The EDF+SDL methodology is very well thought-through. Willemsen's thesis is an interesting read for anyone involved in the field of VR-modeling, as it introduces very simple yet complete constructs for modeling a road network, and thus provides an insight into the feasibility of conceptual VR modeling. However, it seems to have one fatal flaw with regards to the work described in this thesis : the methodology, while very expressive for urban traffic environments, is completely unable to specify anything else. In other words, while it can be used to model any sort of road plan, it is useless in the more general case. This is logical, of course, as it was never intended to be used for anything else.

However, continuing this train of thought, a new methodology would have to be designed for every domain one might want to model in VR. If a battlefield should be designed, a battlefield methodology should be created. If a building is required, a building methodology is needed. Not only would this lead to an enormous amount of time required to develop all these methodologies, but it would also entail problems when several methodologies should be combined. For instance, what if one would want to model a building on a battlefield? Because the focus of this thesis lies on creating a more general modeling technique, for *any* virtual environment, Willemsen's work is largely disjoint from it.

Additionally, while SDL is a scripting language, our scenario modeling language should be graphical, to be more intuitive for non-programmers. Although scripting might not be entirely avoidable, it should be circumvented as much as possible.

1.4.3. SBSD

1.4.3.1. Description

Similar to SDL (section *1.3.2*), Simulation Behavior Specification Diagrams (SBSD), presented in the master's thesis of Carolyn R. Bartley [3], is a scenario description language tailored towards one particular type of VE : in this case, military mission simulations. However, unlike SDL, it is a visual language.

SBSD diagrams are constructed using nodes, representing (sequences of) activities, and transitions between them. All in all, there are two types of nodes, and four types of transitions (*figure 4*).

The node types are (*fig 4a*) :

- ***Atomic Node :*** Represents a single activity performed or assigned to an entity in a simulation scenario.
- ***Multi-task Node :*** Represents a sequence of atomic and multi-task nodes connected by regular and conditional transitions (see below). Multi-task nodes can be expanded to show their lower level build-up.

The transition types are (*fig 4b*) :

- ***Regular Transition :*** The transition is based on the action being executed, in other words, the trigger is incorporated in the implementation of the action itself. Usually, this simply means the transition fires at completion of the action.
- ***Conditional Transition :*** Identical to a regular transition, but with a condition which needs to be satisfied for the transition to take place.
- ***Temporary Reaction Transition :*** Transitions which fire as a reaction to a certain event in the VE. After completion of the new action, execution returns to the action which was being executed previously.
- ***Permanent Reaction Transition :*** Similar to a temporary reaction transition, but execution does not return to the previous node upon completion of the new action.

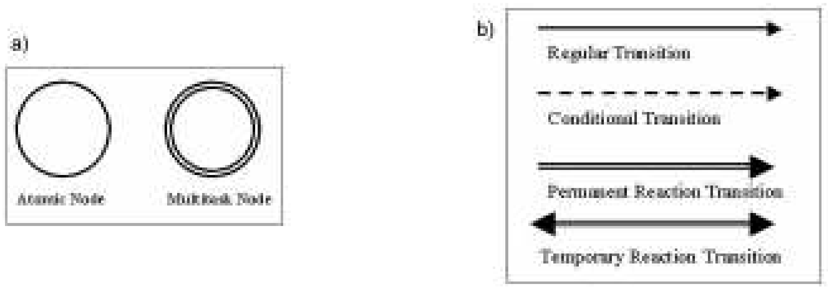


Figure 4 – (a) SBSD nodes; (b) SBSD transitions

Figure 5 provides an example scenario, modeled using SBSD.

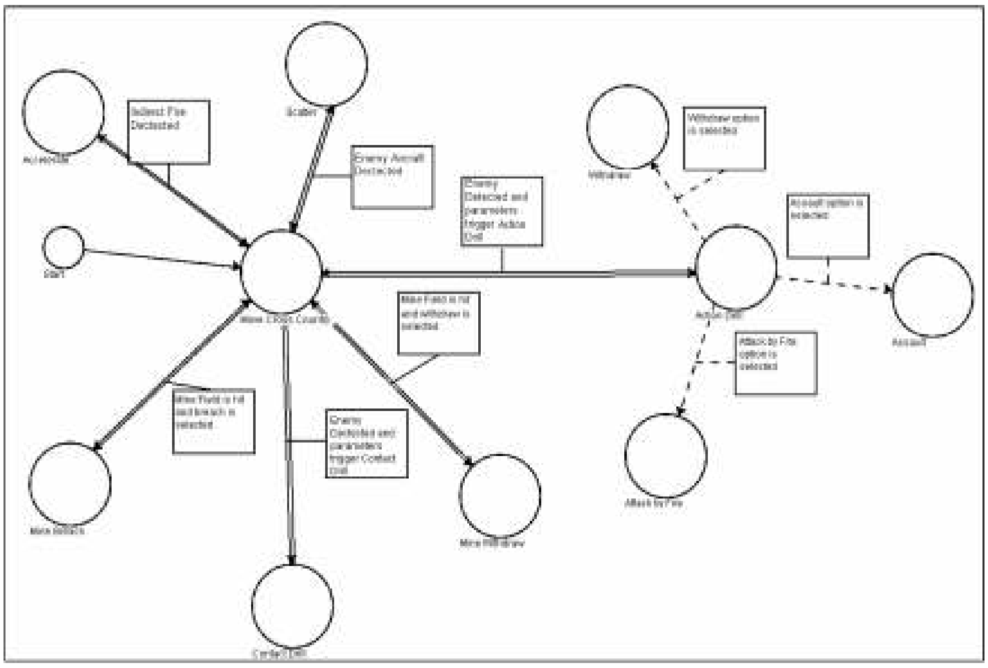

Indirect Fire Detected
Scatter
Enemy Aircraft Detected
Enemy Detected and parameters trigger Action Drill
Withdraw option is selected
Withdraw
Assault option is selected
Start
Move Cross Country
Action Drill
Assault
Mine Field is hit and withdraw is selected
Attack by Fire option is selected
Mine Field is hit and breach is selected
Enemy Detected and parameters trigger Contact Drill
Mine Breach
Attack by Fire
Mine Withdraw
Contact Drill

Figure 5 – Example diagram for a travelling overwatch task

1.4.3.2. Relevance

While SBSD has some interesting properties (which are discussed below), it is not really used for describing full scenarios. One SBSD graph always models behavior for a single object, based on its mission and the way it should react to other entities. In other words, what is modeled is in a sense the AI of the object, so simulations can be run by releasing a certain number of objects in the VE. However, there is no way to model a covering sequence of events that describe a scenario. For example, while it is possible to define how an attack jet should react when its wingman is shot down, it is impossible to declare *that* it is shot down. This is simply a situation which might occur during the execution of the simulation, beyond the control of the designer. Additionally, SBSD lacks a way to specify interaction with the scene.

Therefore, SBSD is incomplete – and not intended – as a full scenario modeling language. It is more akin to the behavior modeling in VR-DeMo, or the actor graphs described in this thesis (chapter 4).

Still, SBSD contains some interesting ideas for modeling scenarios. For one, the notation with linked actions is very intuitive. Furthermore, the different types of transitions provide an insight into possible pitfalls when designing a scenario modeling language : it should be feasible to model behavior interruptions, and transitions based on events occurring in the environment. The notation, as shown in *figure 5*, also supplies an interesting notion, although a negative one : excessive labeling of transitions leads to needless cluttering of the diagram. Hence, it seems wise to attempt to keep transitions free from text as much as possible.

1.4.4. Q

1.4.4.1. Description

The Q [17] language, an extension to Scheme (a Lisp dialect), is a scripting language for defining scenarios of interacting agents. It is based on cues and actions, the former being events that trigger interaction, while the latter are interactions between an agent and the environment. Additionally, guarded commands can be used, where based on a certain cue, a series of actions is performed.

Scenarios in Q are defined as state machines. States are implemented as guarded commands – e.g., if execution is within a certain state, the actions of that state are performed. An example (partial) scenario is given below.

```
(defscenario reception (msg)
  (scene1
    ((?hear "Hello" :from $x)
     (!speak "Hello" :to $x)
     (go scene2))
    ((?hear "Bye")
     (go scene3)))
  (scene2
    ((?hear "Hello" :from $x)
     (!speak "Yes, may I help you?" :to $x))
    (otherwise (go scene3)))
  (scene3 ...))
```

This scenario contains three states : *scene1*, *scene2*, and *scene3*. In *scene1*, the agent will reply “Hello” to anyone who greets him with the same message. After this, he switches to *state2*.

An important aspect of Q is that the implementation of an agent’s cues and actions is unknown to the scenario. They can be implemented by a different person than the scenario designer, as long as both agree on the vocabulary needed by the scenario. To aid the scenario designer (who is not usually a programmer) in creating the scenarios, interaction pattern cards (IPCs) are introduced, a spreadsheet-like card where the designer enters parameters, based on which the scenario’s script is generated. Unfortunately, like the agent vocabulary (e.g. cues and actions), IPCs are domain-dependent, implying that a one or more should be created for each new VE wherein scenarios are executed.

1.4.4.2. Relevance

Again, like in alVRed (*1.4.1*) and SBSD (*1.4.3*), the notion of finite state machines (FSMs) to model scenarios returns. It is becoming apparent that FSMs are a favored tool among scenario researchers.

With regard to the work performed in this thesis, the Q language suffers from the same problem as SDL (section *1.4.2*) : rather than a graphical language, it is a scripting language. Besides this rather obvious complaint, the similarity to SBSD should be noted. In both languages, a scenario is modeled for a single actor, and there is a lack of a higher-level notation to specify the possible courses of a scenario with many interacting actors. Also, like SBSD, Q lacks a clearly defined way to interact with the environment.

1.4.5. PAR

1.4.5.1. Description

Though the Parameterized Action Representation (PAR) language [1] is not really a scenario description language, it contains some interesting characteristics which justify it being incorporated here.

The PAR language is a scripting language for making agents respond to verbal(ized) commands in a context-sensitive fashion. A single PAR is an instantiatable description of an action, corresponding to a certain command. For example, one could define a PAR for the commando “walk around the room”, describing what it means for the agent to walk around the room. In this respect, the PAR language is more of a behavior definition language than a scenario definition language.

Among other things, a PAR contains the following terminology :

- ***Physical Objects :*** The objects referred to within the PAR.
- ***Agent :*** The agent executing the action.
- ***Applicability Conditions :*** Preconditions which need to be satisfied for the action to be executable.
- ***Preparatory Specifications :*** A list of <*Condition, Action*> pairs. Before performing the execution steps (see below), this list is evaluated, and if any of the conditions are false, the corresponding actions are executed.
- ***Execution Steps :*** The actual action being taken, possibly consisting of a number of nested PARs.

- ***Termination Conditions :*** Conditions that, when satisfied, complete the action.
- ***Post Assertions :*** Statements that are executed after the termination conditions have been satisfied.

1.4.5.2. Relevance

Since a scenario will likely be composed of a web of linked actions, the PAR language imparts a few useful insights, particularly the usefulness of including preconditions and results when modeling an action. In addition, like the behavior modeling in VR-DeMo, PAR distinguishes between action definitions and instances. This is something that would undoubtedly prove useful if it could be incorporated in a scenario modeling language.

Another interesting property of the PAR language is the coupling between verbal commands and PAR definitions. While the PAR definitions are (obviously) formally specified – using a Lisp-like notation – they are triggered using natural language commands. Using such a link between easily interpretable commands and intricately defined actions greatly improves the understandability of the language, while retaining expressiveness. This approach should be investigated for use in VR-SDL as well.

1.4.6. Action Frame Based Scenario Description Language

1.4.6.1. Description

The nameless scenario description language designed by Atsushi Ohnishi and Colin Potts [19] is an informal way of specifying scenarios using structured natural language, reminiscent of UML use cases. Because of this, it is very intuitive to use. An example is given below.

```
1.  Passenger is at floor 2.
2.  Passenger decides to go to Floor 3.
3.  Passenger enters a car.
4.  Passenger pushes Button[3]
5.  Car feedback to Passenger the status(going to 3) by
    lighting the Dir[UP] and Button[3] on
6.  Car starts UP
7.  Car arrives at 3 and stops
8.  Car feedbacks to Passenger the status(at 3) by
    lighting DIR[UP] and Button[3] off and ringing the
    bell
9.  Doors open
10. Passenger gets off the car
```

Moreover, a number of action frames are defined, which are formally defined event templates. Each of the informally specified events in the example above can be transformed into a formal action frame. Unfortunately, no rules are given for this, so the designer basically needs to redefine his scenario using the more formal syntax.

1.4.6.2. Relevance

Instead of modeling all possible courses of a scenario, the action frame scenario language only models a single execution of it. For the research in this thesis, this is inadequate, as it should be able to describe a branching scenario. For example, using Ohnishi's language, one is unable to specify what happens if the passenger pushes a different button than 3. To model this, an entirely new scenario would have to be created, completely analogous to the example, but with a different button number. This would quite obviously lead to an enormous overhead.

One thing about Ohnishi's language that is remarkable, is the intuitiveness of the (informal) notation. When modeling actions, a similar notation, using roughly an *<actor><action><object>* syntax, where the actor performs the action on the object, would probably result in a comparably instinctive feel.

1.5. Conclusion

Even though plenty of research has been performed on the general subject of scenario and behavior modeling, it is clear that the exact subject of this thesis remains largely uncultivated. Most 'scenario' description languages face one or several of the following problems :

- Rather than graphical languages, they are scripting languages.
- They are too informal to retain any prospect of code generation.
- Interaction with the environment is not – or too poorly – described.
- They are too low level, over-focused on a single actor, and thus, more directed at behavior modeling than scenario modeling.
- They are designed for a specific purpose (urban traffic, military operation...), and are too low-level to be used for modeling of scenarios in a general VE.

While this certainly does not imply that these languages are bad, it does indicate that they were designed with a different perspective in mind than the work performed here. Regardless, many interesting ideas for general scenario modeling were extracted, and put into practice while designing VR-SDL.

Chapter 2

VR-DeMo : An overview

2.1. Introduction

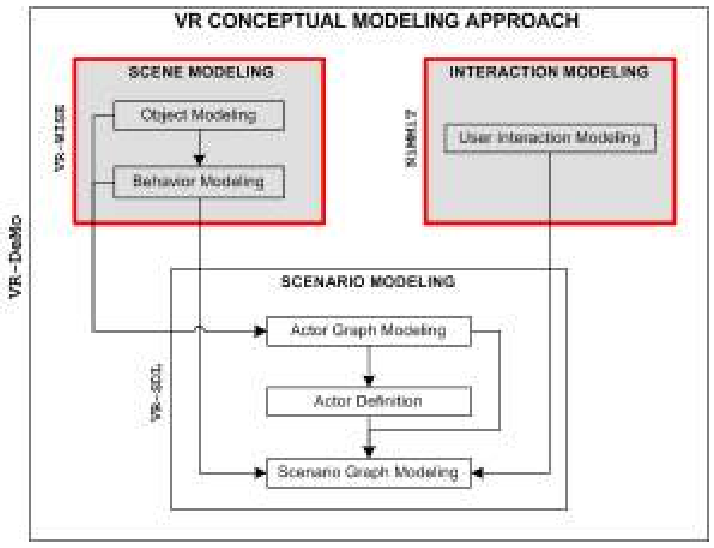


Figure 6 – Scope of this chapter : *scene* and *interaction modeling*

Earlier in this thesis, the need for a conceptual modeling approach for VR was established. VR-DeMo is the name of a project, financed by the *Flemish Subsidy Organization* (IWT), which aims to achieve exactly this. Two research partners, the *Universiteit Hasselt* (Uhasselt) and the *Vrije Universiteit Brussel* (VUB) are cooperating in attempting to create such an approach. In general, the objectives[3] of the project are :

1. The definition of high-level modeling concepts that allow for the description of the objects in a virtual environment at a conceptual level.
2. The definition of high-level modeling concepts that allow for the description of the behavior of objects in a virtual environment at a conceptual level.
3. The definition of modeling concepts that allow for the description of rules and constraints in a virtual environment.

[3] as specified on *http://www2.edm.uhasselt.be/vr-demo/* (access date 5/5/2007)

4. The definition of modeling concepts that allow for the description of the individual modalities for interaction with a virtual environment.
5. The definition of modeling concepts that allow for the description of multi-modal interaction with a virtual environment.
6. To couple this conceptual modeling concepts to low level, implementation concepts for a particular VR-technology.
7. The design and development of a common test case that integrates the results of all work packages.

The part of the research being conducted at the VUB is performed by the *Web & Information Systems Engineering* (WISE) research group, and is aptly named VR-WISE. With regards to this thesis, it is the most important part of the VR-DeMo project, as it is most closely tied to the VR-SDL language for modeling scenarios in VR.

As shown in *figure 6*, VR-WISE is mainly concerned with objectives 1, 2 and 3, creating a conceptual modeling technique for objects (both simple and complex) within the VE, and for the behaviors linked to these objects.

The work performed at the *Universiteit Hasselt* is mostly concerned with user interaction with the VE, through interfaces and menus. In a sense, this can be seen as meta-level interaction, since the user does not manipulate the world directly, but uses concepts from outside the VE (menus, widgets, pointers...) to influence it. An approach called NiMMiT was developed for modeling this. Its position in the general VR conceptual modeling process is demonstrated in *figure 6*.

Since the aim is to model scenarios involving the lower-level objects, there is a very strong coupling between the (high-level) *scenario modeling*, and the (lower-level) *object* and *behavior modeling*. However, since the actions of the user might influence the course of a scenario, there is some affinity with *interaction modeling* as well.

In the remainder of this chapter, a brief overview of VR-DeMo (mostly focused on VR-WISE) will be given, focusing on the research described in part 2 of the thesis. This serves a dual purpose : firstly, to supply the reader with a background against which to situate the work performed, and secondly, to ensure a better understanding of VR-SDL. The reader already familiar with VR-WISE may skip this chapter, but it is strongly recommended that others read at least sections *2.2* and *2.6*. For a full comprehension of the *scenario modeling* approach (and particularly the *actor graph modeling* described in chapter 4), knowledge of the constructs from sections *2.3* and *2.4* is required.

The chapter is structured as follows : initially, the general idea behind VR-WISE is described, and consequently, the *object* and *behavior modeling* phases are explained in more detail. Both phases are subdivided in two sections, explaining simple and more advanced modeling, respectively. Finally, a brief overview of the NiMMiT approach is given as well. The attention spent on both topics is proportionate to the tightness of their relation with VR-SDL.

2.2. VR-WISE : General Approach

The design process for conceptual modeling of a VE using the VR-WISE approach [21], consists of three steps : the ***specification step***, the ***mapping step*** and the ***generation step*** (*figure 7*). Each of these steps uses ontologies [18] as an underlying representation formalism. Ontologies in VR-WISE are employed for two different purposes : to explicitly represent the knowledge about the domain of the VE (e.g. traffic), and to internally define the available modeling concepts, and relations between them. In other words, ontologies are both used to represent knowledge about *what* is modeled, and to specify *how* to model things (meta-level).

Below, the three steps in the design process, and the corresponding ontologies, are outlined.

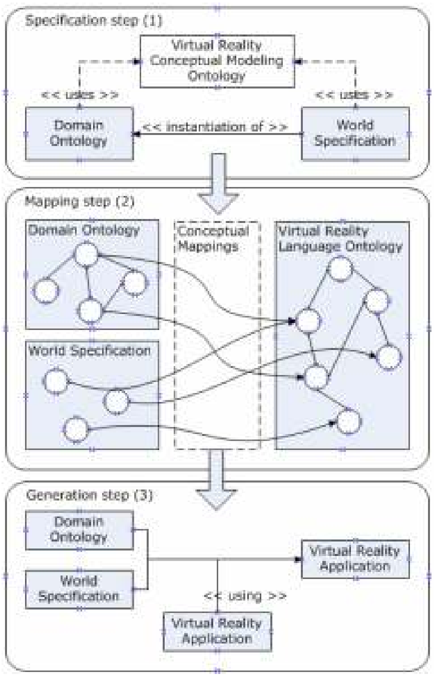


Figure 7 – VR-WISE approach

The **specification step** consists of the highest-level modeling. In this step, the designer specifies the structure of the VE using domain knowledge, without describing implementation details (such as the representation of the concepts used in the VE). This step uses three ontologies :

- The ***Domain Ontology*** describes the concepts used in the VE's domain (much like object types in OO-programming), their properties and the relations between them. For example, in the traffic domain, this ontology would include concepts such as *Car*, *Pedestrian*, *Street*, *Crossroad*, *Building*, and relationships like *"a Car drives on a Street"* or *"a Crossroad connects several Streets"*. If possible, it is of course a good idea to use an existing ontology for this, rather than creating one from scratch.
- The ***World Specification*** contains the actual conceptual description of the VE to be built. It is created by instantiating the concepts from the domain ontology, and adding instance-specific information (such as color, size, location...) and world-specific information (gravity, lighting...). In the traffic example, this means that, instead of saying there are "cars" in the world, one would specify the kind of cars (the make) and how many of them there are.
- The ***Virtual Reality Conceptual Modeling Ontology*** is a meta-level ontology, describing the constructs available for modeling the VE (concepts, instances, properties...). Both the Domain Ontology and the World Specification are built using the specification defined in this ontology. It contains a number of sub-ontologies, each describing a certain aspect of the modeling process (object modeling, behavior modeling...)

The **mapping step** provides a connection between the conceptual level and the implementation level. Like the specification step, it uses three different ontologies.

- The ***Virtual Reality Language Ontology*** defines a number of low-level VR primitives which can be used for the implementation of the higher level concepts. Examples include geometrical items like spheres or cubes. Note that, contrary to the VR Conceptual Modeling Ontology from the previous step, this is not a meta-level ontology : it specifies items usable for modeling (*what*), rather than the way to use them (*how*).
- The ***Domain Mapping*** defines mappings between the high-level modeling concepts from the Domain Ontology in step 1, and the primitives specified in the VR Language Ontology. For example, in the traffic example, suppose the application built has as its sole purpose to study mass-behavior, then it would be wise to choose very abstract representations for

the different concepts, to avoid wasting computing power on rendering the objects' avatars. Hence, a pedestrian could for example be mapped to a cone, and a car to a prism. Obviously, more elaborate mappings are also allowed, should the focus of the application lie more on the visual side of things.

- The ***World Mapping*** is entirely similar to the Domain Mapping, with the obvious exception that it maps items from the specification step's World Specification to primitives from the VR Language Ontology. This is useful to designate a different representation to certain instances : while a Lamborghini Gallardo and a Volkswagen Beetle are both cars, they clearly do not look the same, and thus, require different mappings.

The **generation step** is the final act in the design process. In this stage, the definition of the VE given by the Domain and World Ontologies, are converted into a working application, using the corresponding mappings.

With this methodology in mind, it is clear that the *specification step* is of the greatest interest to this thesis. After all, it is the modeling techniques used in this step that will be extended to include a scenario description language. To fully comprehend the extension, it is of course imperative to grasp the basis on which it is built. Therefore, the next two sections will explain the two parts of the modeling technique currently present in VR-WISE : *object modeling*, and *behavior modeling*. Recall that both modeling approaches employ a graphical notation, for maximum intuitiveness amongst non-programmers.

2.3. Object Modeling in VR-WISE

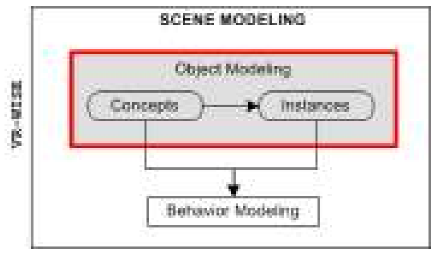


Figure 8 – Scope of this section : *object modeling*

As its name suggests, the object modeling phase consists of modeling the objects populating the virtual world. Two important issues are tackled : firstly, the objects' internal structure should be defined, and secondly, their positioning within the virtual world should be specified.

2.3.1. Simple Objects

2.3.1.1. Internal structure

In many ways, the approach taken to object modeling in VR-WISE is reminiscent of objects in an object-oriented programming language. The different types of objects in the world are modeled as concepts (analogous to OO classes), which are then instantiated into the entities populating the VE (*figure 8*).

Concepts represent the object types at domain level. Similar to data fields in OO classes, they may contain a number of properties, each of which might have a default value assigned to it. There are two different types of properties :

- ***Visual properties*** represent visual information about the object, like its geometrical representation (limited to simple shapes, like spheres), colors and / or texture.
- ***Non-visual properties*** can contain information like the object's weight, name, mass, etc...

Visually, concepts are represented by a rectangle containing the concept name, and a drop-down box detailing the properties with their default values. This is illustrated in *figure 9.*

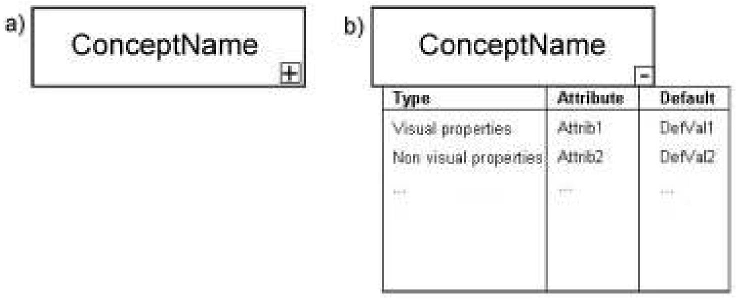


Figure 9 – (a) Simple concept notation; (b) Detailed concept notation, with the property drop-down box shown

Instances are the actual objects populating the VE. They share the same characteristics as the concept they instantiate, though some of the default property values may be overwritten. The instance notation, as shown in *figure 10*, is an ellipse, containing a similar drop-down box as the concept notation.

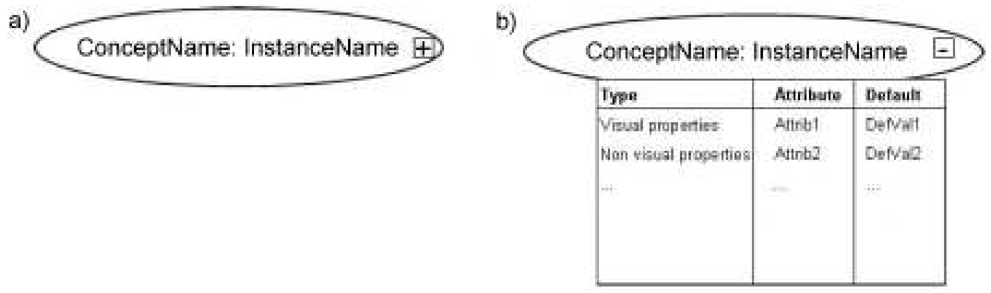


Figure 10 – (a) Simple instance notation; (b) Detailed instance notation, with the property drop-down box shown

2.3.1.2. Spatial Relations

Aside from specifying the outlook and properties of an object, it should also be placed in the world. To avoid having to enter low-level coordinates, a graphical notation has been developed for denoting spatial relations between objects. Basically, three types of relationships are present :

- ***Directional relationships*** express an object's position in terms of another one, using concepts such as *left*, *right*, *above*, *under*, *front*, and *behind*.
- ***Orientation relationships*** are concerned with the rotation of an object. Object's are given an internal coordinate system defining their front, back and sides, and consequently, this can be used to determine the orientation of an object in terms of another one.
- ***Metric relationships*** specify distances. They are often combined with directional relationships, to express things like "the chair is 3 meters left of the table".

Figure 11 provides an example of the notations used to express positioning. It describes that instance *A* is positioned 2 meters in front of instance *B* (and not the other way around, mind the arrow), and both instances are facing each other with their front sides.

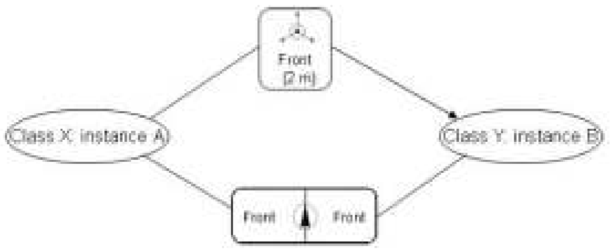


Figure 11 – Example demonstrating the spatial relationship notation

2.3.2. Complex Objects

Since simple objects are limited in their visual display, a new concept is needed to be able to model more elaborate objects : complex objects. These are objects which consist of several subobjects, which are themselves either simple or complex. Complex objects are defined on the concept level, and like simple objects, they should be instantiated before they can populate the VE. Instantiating a complex object is straightforward : one should merely instantiate all of its subobjects.

2.3.2.1. Connecting Objects

Naturally, when combining objects, it is imperative to specify *how* they are combined. Therefore, three connection relations are defined :

- ***Connection point :*** using this techniques, the connected objects share a single point. This is useful to model ball and socket joints, like human shoulders.
- ***Connection axis :*** both objects share a single axis. An obvious application of this type of connection is a hinge joint (e.g. a human knee).
- ***Connection surface :*** the objects are connected through a common surface. This implies a rigid connection, and thus, it is most useful for creating immovable connections between objects.

The graphical notation for connection relations is similar to that of directional relationships. *Figure 12* gives an example for the case of a connection point, modeling a door by connecting two simple objects : a door board and a door handle. Note the specification of a spatial relationship between the two concepts (the top rectangle), to indicate the planes in which the connection point will lie. The bottom-most rectangle describes the actual connection relation. It is shifted 0.4 meters to the right of its initial position (the centre of the door board).

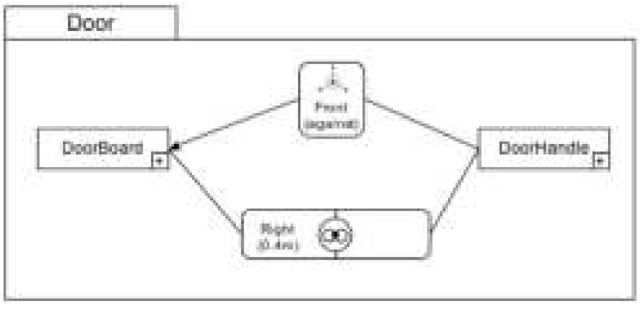


Figure 12 – Example demonstrating the connection point notation

Connection axes and surfaces use a completely analogous notation, the sole variation being the use of a different symbol in the connection relation box.

2.3.2.2. Roles

It is easily imaginable that one would want to create a complex object where a single concept is used for multiple purposes. For example, consider a table, consisting of a top with four legs : each of these legs would be modeled in the exact same way, suggesting the legs would be members of the same concept (e.g. *TableLeg*). However, merely making four separate connections between the *TableTop* and *TableLeg* concepts would entail the issue of the legs not being mutually distinguishable.

Therefore, ***roles*** are introduced. When multiple objects of the same concept are used in the construction of a complex object, each object is labeled with a role name. Roles are graphically denoted by a double-edged rectangle with the role name, and the concept name below it, between "< >". *Figure 13* illustrates this by means of the table example.

Roles are an important concept from the perspective of the scenario modeling, since objects participating in the scenario might be complex, in which case the role names are the way to reference their subobjects.

Note that *figure 13* provides a definition of a complex object *Table*, it does not instantiate it. As mentioned at the start of section *2.3.2*, instantiating a complex object concerns instantiating all of its subobjects. This is demonstrated in *figure 14*, by instantiating the table object.

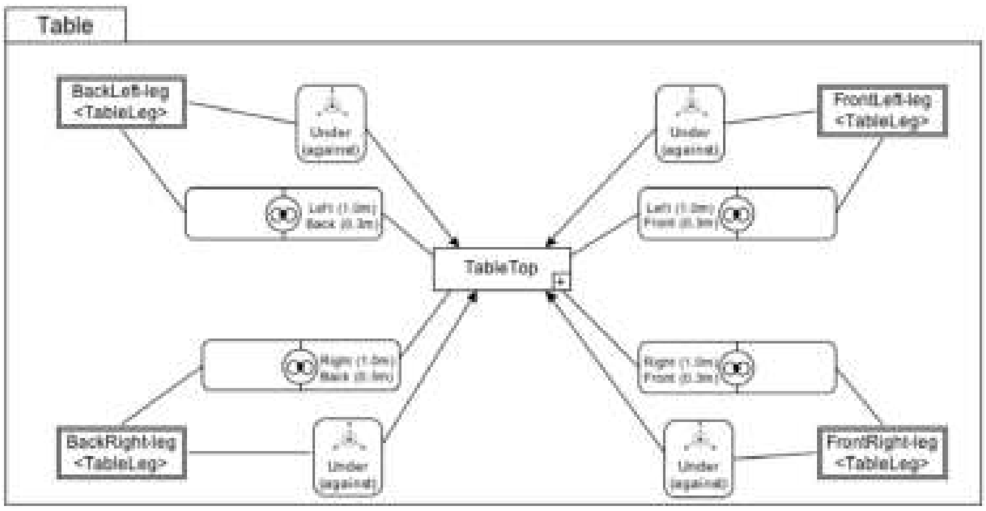


Figure 13 – Role example : definition of a table

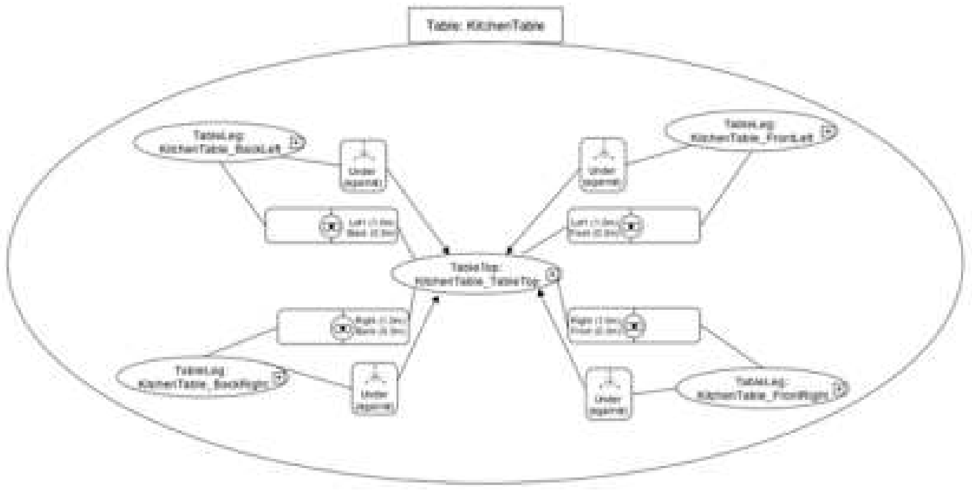


Table: KitchenTable
TableLeg: KitchenTable_BackLeft
TableLeg: KitchenTable_FrontLeft
Under (against)
Under (against)
Left (1.0m) Back (0.5m)
Left (1.0m) Front (0.5m)
TableTop: KitchenTable_TableTop
Right (1.0m) Back (0.5m)
Right (1.0m) Front (0.5m)
TableLeg: KitchenTable_BackRight
Under (against)
Under (against)
TableLeg: KitchenTable_FrontRight

Figure 14 – Complex object instance notation

2.3.2.3. Unconnected Complex Objects

The description provided so far in section *2.3.2* was concerned with modeling connected complex objects; objects consisting of several simpler objects glued together, much like a *Lego* car is created by piling up blocks.

However, consider the example of a soccer team : while the objects of the team (the players) are clearly stand-alone, it does in some respects make sense to consider the team as a whole. Tactics can be executed in which all the players of the team should cooperate, and properties can be thought up, referring to the entire team, instead of individual players – for example, the time they have been playing together, a property which could influence the smoothness of execution in the team’s play.

Therefore, it is allowed to define unconnected complex objects : these are, as the name suggests, objects whose subobjects are not physically connected to one another, but do belong together conceptually. The notation is completely similar to that of (connected) complex objects, except that the surrounding rectangle is drawn with a dashed border, and that no connection relations can be defined. It is possible to describe spatial relations, however, signifying the initial position of the subobjects. In the soccer team, for instance, this could signify a 4-4-2 or 4-3-3 positioning at the start of the game. Naturally, when the game is in session, the players would move around freely and break this initial positioning.

2.4. Behavior Modeling in VR-WISE

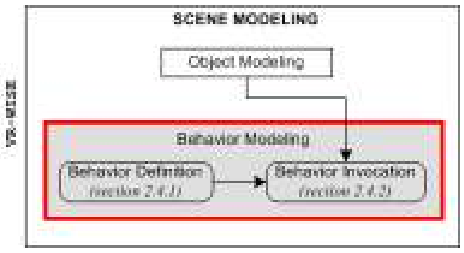


Figure 15 – Scope of this section : *behavior modeling*

The object modeling technique, as described in section *2.3*, provides the designer with the means to design an attractive virtual world, filled with a variety of objects. However, as it stands, there is no way to attach behaviors to the objects, prompting them to stand idly. Therefore, in order to increase the realism, and provide the VE with a bustling, vivid atmosphere, a behavior modeling technique is present in VR-WISE.

As displayed in *figure 15*, modeling a behavior involves two different kinds of diagrams : ***behavior definition diagrams***, and ***behavior invocation diagrams***. Behavior definition diagrams contain the general specification of the behavior, loose from the actual objects populating the VE. Behavior invocation diagrams consequently couple the defined behaviors to either concepts or instances, thus inserting them into the virtual world – in a sense, this can be viewed as instantiating the behavior.

2.4.1. Behavior Definition Diagram

2.4.1.1. Actors

In a behavior definition diagram, an abstract concept called ***actors*** is introduced. Behaviors are defined in terms of these actors, thus providing the ability to reuse them by linking the actors to different objects at behavior invocation.

Actors are similar to interfaces in Java or C++. They can contain a number of properties, which objects playing the role of a certain actor must also posses. *Figure 16* provides an example of the actor notation.

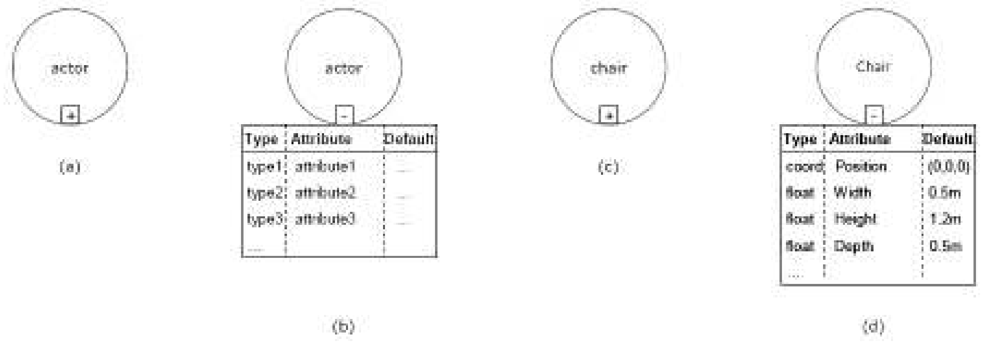


Figure 16 – Actor notation

Figure 16 a and b show an actor, respectively with properties hidden and shown. *Figure 16* c and d show a practical example, a chair actor with properties *Position*, *Width*, *Height* and *Depth*. Any object able to play the role of the chair actor in its corresponding behavior should contain at least these properties.

Additionally, an inheritance mechanism is in place to allow for the declaration of parent and child actors. Child actors inherit all of the properties and behaviors of their parent actor, but may add new ones or overwrite some of them.

2.4.1.2. Simple and Complex Behavior

Knowing how to specify the actors participating in a behavior, the next step is to model the behavior itself. An overview of the approach will be given in this section, but since it can be seen as a black box to the rest of the thesis, it will be kept brief.

Behaviors are defined as either primitive or complex. Primitive behaviors consist of actions like move, turn, roll, or mechanical behaviors, defining movements over a connection point or axis in a complex object. To form complex behavior, other behaviors (either primitive or complex) are combined using operators. There are ***temporal operators*** (e.g. behavior A is executed before behavior B), ***life time operators*** (e.g. behavior A is suspended for some time and resumed later) or ***conditional operators*** (e.g. behavior A is executed if a certain condition is true).

Graphically, both primitive and complex behaviors are represented by a rectangle. The top-level behavior has its actors linked to it. Note that within the definition of a behavior, the actor's parameters are accessible, and can be manipulated. Two behaviors are connected by an operator by linking their representations using an arrow with the operator notation (a rounded rectangle containing the operator's symbol) on it.

The way in which actors are linked to a behavior deserves special notice : an actor can either be actively participating in a behavior, or passively undergoing it. In this second case, the actor (called a ***reference actor***) can be used to specify some primitive behavior in terms of it. Regular actors are linked to behaviors using a line, reference actors are linked with a dashed arrow. To clarify the concept of a reference actor, assume one would like to model a behavior where an actor points at some object. Obviously, to know where to point, the location of the target object should be known, and thus, it should be linked to the behavior as a reference object.

Figure 17 displays a behavior definition diagram, in the context of a robot in a manufacturing plant. Rather than as a comprehensive example, it is merely intended as an illustration of the notations used.

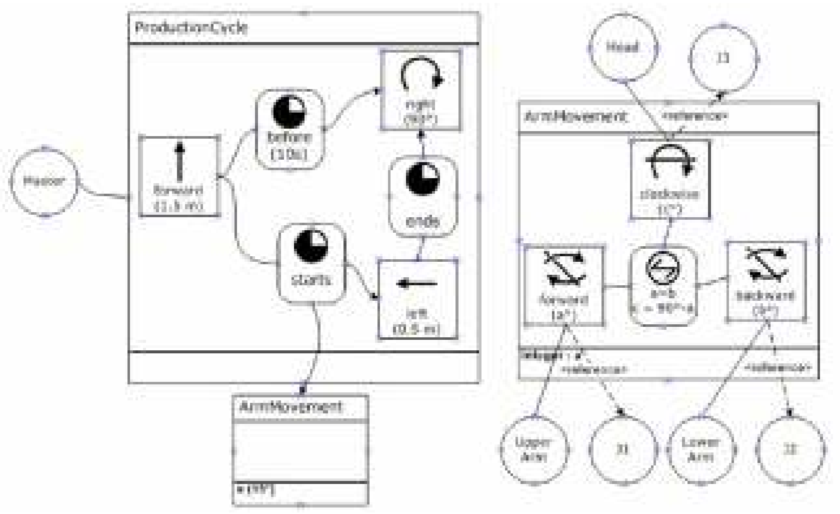


Figure 17 – Behavior definition diagram example

There are a number of more complex behavioral concepts, but these fall outside the scope of this thesis. Refer to [21] for a more comprehensive overview of behavior modeling in VR-WISE.

2.4.2. Behavior Invocation Diagram

2.4.2.1. Instances

As mentioned before, simply defining the behavior is obviously insufficient. For it to be incorporated in the VE, its actors should be linked to actual objects in the world. This is called invoking the behaviors. It is achieved by using behavior invocation diagrams.

Behavior invocation diagrams do little more than linking the abstract actors to concrete objects from the VE. Basically, there are two ways of invoking a behavior : either the actors can be linked to a concept, or to an instance. In the first case, all instances of the concept will exhibit the behavior. In the second case, only the particular instance will. Both concepts and instances are denoted as described in section *2.3.1.1* : concepts as rectangles, instances as ellipses. As mentioned earlier, for a concept or instance to be eligible to be linked to an actor, it must contain the same properties as the actor.

Besides actor instantiations, behavior invocation diagrams are linked to events, specifying what triggers them : a mouse click, an interaction with some other object, a certain amount of time passed, and so on. Events are represented by a hexagon.

Figure 18 demonstrates the invocation notation, by displaying a behavior invocation diagram compatible with the definition diagram shown in *figure 17*.

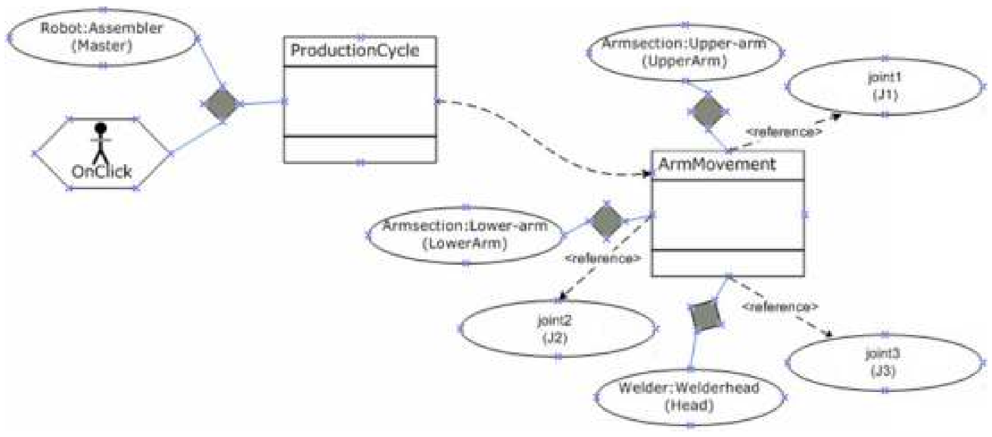


Figure 18 – Behavior invocation diagram example

2.5. NiMMiT : Modeling User Interaction

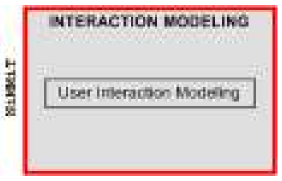


Figure 19 – Scope of this section : *interaction modeling*

So far, everything discussed in this chapter was directly related to modeling the contents of a VE. However, to model a VR application, merely constructing a virtual environment is insufficient : the user should be able to interact with the environment, through some interface. NiMMiT [29] is a notation for modeling user interaction in VR (*figure 19*). It is a low-level technique, used to specify the actions a user should take through input devices, in order to perform tasks in a VE. *Figure 20* illustrates the NiMMiT notation, by providing a diagram for selecting an object in the VE. This is achieved by moving a pointer over the object, at which point it will be highlighted, and clicking to select it.

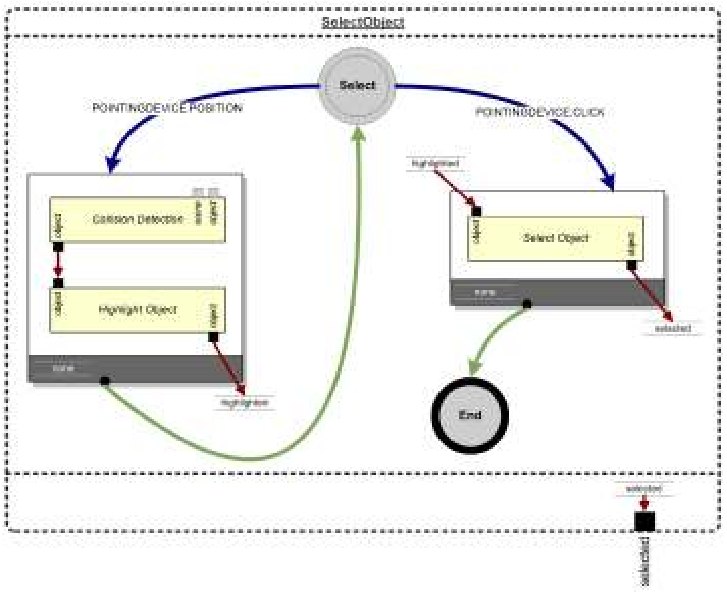


Figure 20 – A NiMMiT diagram modeling the selection of an object in a VE

The main components of a NiMMiT diagram are :

- ***States and Events :*** States, drawn as circles, represent the different stages an interaction can be in. *Figure 20* contains only two states, *Select* and *End*. Events are actions taken by the user (or more specifically, by input devices), which cause task chains to fire. They are drawn as arrows, labeled with the event name. In the example, the blue arrows are events.
- ***Task chains and Tasks :*** A task chains is a linear succession of tasks, drawn as a white box with a grey border. Each task chain contains at least one task (depicted as yellow rectangles), representing actions to be taken by the user. A number of predefined tasks exist (e.g. selecting, moving or deleting objects), but it is also possible to define new tasks using a scripting language, or to hierarchically use an entire interaction diagram as a task.

- ***Data flow, Data types and Labels :*** NiMMiT diagrams contain constructs for handling data : a task may have several input and output ports, denoting data required and returned by the task, respectively. The shape of the port (in *figure 20*, all ports are shown as black squares) indicates the data type. Output ports of a task may be connected to input ports of a follow-up task in the chain, but only if they are of the same type. Evidently, connecting two ports means the output from the former task is used as input for the latter. Finally, output data might also be placed in labels, variables to allow transition of data between different task chains. Labels are drawn as small boxes, connected to an input or output port.
- ***State transition and Conditional state transition :*** After successful execution of a task chain, a state transition (drawn as a green arrow in the example) takes place. State transitions may be conditional, in which case the next state is dependent on the value of some label.

Using these components, NiMMiT diagrams can be constructed that are both event, state, and data driven.

2.6. Conclusion

In this chapter, the parts of the VR-DeMo approach relevant to this thesis were explained. VR-DeMo is a project aimed at creating a conceptual modeling technique for virtual environments. From the perspective of scenario modeling, the most important part of VR-DeMo is VR-WISE, the technique for modeling the objects populating the virtual environment, and the behaviors they execute. Both the object and behavior modeling phases were described in this section, giving the reader an initial feel for the concepts and notation behind the approach.

Additionally, a concise overview of NiMMiT was provided, the technique employed by VR-DeMo for modeling user interaction. Though not as closely tied to scenario modeling as VR-WISE, it is still linked to it.

In the next part of the thesis, the VR-DeMo approach will be extended with VR-SDL, a notation for modeling scenarios for virtual environments.

Part II

Modeling Scenarios in VR-DeMo

Chapter 3

Methodology

3.1. Introduction

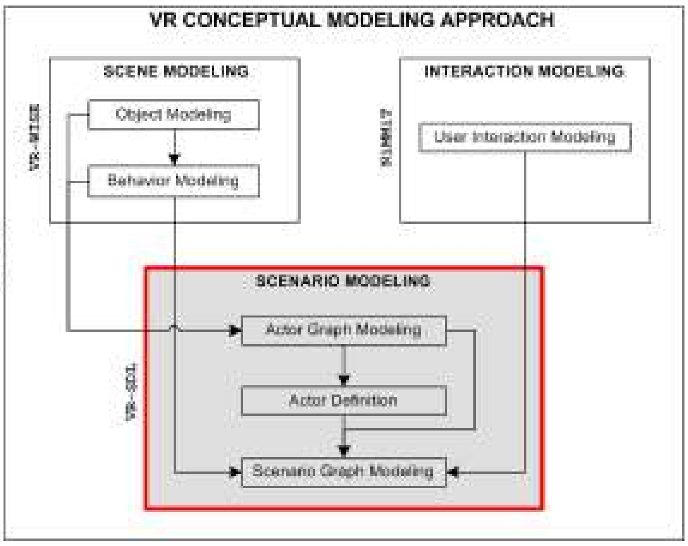


Figure 21 – Scope of this chapter : *scenario modeling*

In the previous chapter, the current state of the VR-DeMo project for conceptual modeling of virtual reality was described. A number of interesting results have been achieved so far : through VR-WISE, the static scene can be modeled by designing the structure and position of the objects therein, and these objects can be assigned behaviors to make the world feel more vivid and realistic. Additionally, NiMMiT provides a way to model user interaction.

However, there is still an obvious gap in the modeling process. As it stands, modeling an intricately detailed VE with VR-WISE has little purpose, since it is impossible to model the goal of this VE. Elaborate interactions can be specified with NiMMiT, but they have no context to be used in. Therefore, VR-SDL is introduced, a notation for modeling VR scenarios (*figure 21*). Through VR-SDL, the application for which the VE was created may be modeled, and the interactions can be utilized in higher level descriptions.

In part 2 of this thesis, VR-SDL will be presented. However, rather than jumping straight into the deep end of the pool, with the formal specifications of the notation, it is wise to start off slowly, and familiarize the reader with the general methodology of the approach first. Due to the duality of the VR-SDL notation, with two different types of graphs, a failure to include a general introduction might cause the reader to lose track of the larger picture. Hence, this chapter is a recommended read for anyone trying to gain insight in VR-SDL.

The duality mentioned above follows from the fact that in VR-SDL, the actors of the scenario are modeled separately from the scenario itself. When designing a scenario, the first thing to do is to get a global picture of what will happen. Once the designer has this in mind, he must analyze the different types of scenario actors[4] that will be participating in the scenario. Consequently, an actor graph is created for each actor type, detailing the evolution it can undergo during execution of the scenario. After completing this step, the actors of the scenario should be defined formally (their types and names), and only then, everything is available for modeling the scenario. This is done by drawing a scenario graph, describing the possible courses the scenario might take, and the influence this has on the actors.

An additional aim of this section is to demonstrate the practical benefit of a VR scenario modeling language. Therefore, an example of a useful VR application will be provided, modeled using the VR-SDL technique. This also constitutes as a preliminary test for intuitiveness of the notation, since the reader will be confronted with it before having received a detailed explanation.

3.2. Design Guidelines

Before delving deeper into the specifics of the approach, it seems advisable to consider the guidelines that were followed when designing the VR-SDL scenario notation.

First of all, the notation should be formal. Rather than a guideline, this is an actual requirement, since diagrams created with the VR-SDL language should eventually be usable for automatically generating code for the scenario – this can not be done if the notation is informal (using natural language, for example).

[4] Not to be confused with behavior actors, as described in section *2.4.1.1*.

Since the notation should be usable by laymen and professionals alike, the most important concern is that it should be intuitive. This is the reason why a graphical notation is chosen, rather than a scripting language. Of course, badly-designed graphical languages may still be unintuitive, so a few additional points were taken into consideration :

- Inconsistency greatly decreases intuitiveness, so the language should be as consistent as possible. That is, as often as possible, identical constructs should be used for identical concepts.
- Contrarily, using the same notation for different constructs should be avoided as much as possible.
- To further increase consistency, naming conventions should be provided for different constructs. As long as it does not decrease readability, it may not be a bad idea to stay conform to the naming conventions of a popular programming language (*Java*, for example). This would entail an increased familiarity for programmers using the language, while not making the notation less instinctive for non-programmers.
- For ease of learning, it is important that the amount of constructs remains limited, while at the same time providing enough flexibility to model any type of scenario.
- Inevitably, textual notations will be required for some of the modeling. These are kept as brief and simple as possible. In addition, each textual notation is preceded by a keyword, indicating what it describes (e.g. a property, a behavior, etc.).

In addition to intuitiveness, an extra guideline is that the diagrams created with the VR-SDL language should be as compact as possible. Therefore, needlessly elaborate notations should be avoided as much as possible.

3.3. An Example : Room Furnishing

3.3.1. Virtual Reality in Practice

Redecorating a room can be a tedious job. Just coming back from the hardware store, having bought some color paint. Then, after having painted a single wall, you realize the color is too dark after all. Or suppose you have just bought an expensive new table, but upon installing it, you notice that it actually takes up a bit too much space in your living room, and thus, does not look as good as it seemed in the shop.

The problem, of course, is that you have to rely solely on your imagination to picture how things will look, often using incomplete information (like the tiny paint color square). It would obviously help considerably if you could see a representation of what the room would look like, before actually having changed it. This is where VR comes in.

As mentioned in the introduction to this thesis, real estate already uses 3D house models to communicate with customers from time to time : aside from providing the ability to view their prospective new home at any time, and from any angle, it offers homeowners-to-be the ability to customize the appearance of their dream house. In one mouse-click, they can see how the bedroom would look without that antique rug they do not like, or decide whether the living room would look better in light green, or light blue. This is not science fiction : applications such as this actually exist today, like the one described in [16], which is freely downloadable.

Taking things a step further, one could even envision a cooperation with furniture stores, handing out promotional CD-ROMs (or DVDs, if you prefer) with 3D models of their products, which can be placed in a virtual house. Looking even further into the future, advanced user interaction devices like visors and sensor gloves can make the experience so realistic, that it is barely distinguishable from reality.

But for now, let us stick to the present, and take a look at a detailed description of a similar – yet somewhat simplified – application.

3.3.2. A Room Furnishing Application

Consider a simplified version of the application described above. The application allows the user to model and customize a room layout. Note that this application will be used for demonstrating the VR-SDL notation, and is thus quite limited in scope. Nevertheless, while a fully functional, commercial application would include more options, it would not necessarily require more elaborate modeling constructs.

The interface of the application consists of a window displaying the virtual room, and a number of interaction buttons. The user has no physical representation in the VE, he merely uses a point-and-click device (a mouse, in the simplest case) to click the correct buttons or locations in the VE. Customizing a room is done in three steps, which the user can alternate between at will :

- **Pick the shape and size of the room.** The available shapes are a long-stretched rectangular room, a wide rectangular room, a square room, and an L-shaped room. Obviously, more shapes could be added in a later stage. The room shape is altered by clicking the corresponding shape button, and buttons are provided to increase or decrease the diameter by 0.1 meter.
- **Choose the wall color or wallpaper.** Walls can be given a color and a wallpaper, chosen from a number of available motives. Coloring is done by right-clicking when the pointer is aimed at a wall, and selecting the correct color from a drop-down menu. Wallpaper and colors are complementary, one of each can be chosen. For example, if flower wallpaper and a red color are chosen, the flowers on the wallpaper will be red. Both the color and the wallpaper have a default value (*White* and *None*, respectively).
- **Furnish the room.** In this step, a number of actions can be taken to adjust the room furnishing :
    - **Add a piece of furniture.** The user can choose between a chair, bed, desk or cupboard. Furniture is added by clicking the corresponding button, and left-clicking the location in the VE where it should be added.
    - **Select or deselect furniture.** Furniture is selected by clicking on it when it is not selected, and deselected by clicking on it if it is selected. Any amount of furniture can be selected at the same time.
    - **Move furniture.** By dragging the selected furniture, it is moved across the room. Note that if multiple pieces of furniture are selected, they are all moved simultaneously.
    - **Delete the selected furniture.**
    - **Rotate furniture.** By clicking the rotate left or right buttons, all selected furniture is rotated 45 degrees in the selected direction.
    - **Recolor furniture.** Furniture can be recolored or retextured in the same way as walls can. Be advised, however, that unlike for walls, textures and colors are not complementary for furniture : if one is chosen, the other is cancelled out.

Next, let us elaborate on how to model such an application using VR-SDL.

3.3.3. Modeling the Room Furnishing Application

3.3.3.1. Actor Graphs

When considering how to model the *room furnishing application*, a first thing to notice is that the world contains objects that can be manipulated : the room itself, the furniture which is added or altered, ... Since these objects are clearly lower-level than the actual scenario, the modeling of the objects in VR-SDL is separated from the modeling of the actual scenario[5]. Using this approach leads to an increased separation and reusability of the lower-level objects. This is achieved by using actor graphs, an advanced form of state machines tailored specifically towards objects in a scenario. An actor graph models the way an object can evolve over the course of the scenario, by describing different states it can be in, and possible ways to transitions between them. A state may contain various information, such as conditions, behaviors, and visual information.

In VR-SDL, objects participating in a scenario are called actors. Note that this is unrelated to the concept of actors as defined in behavior modeling. In part 2 of the thesis, whenever actors are mentioned, it will always refer to actors within a scenario (unless explicitly stated otherwise).

Returning to the room furnishing example, it would seem from the description in the previous section that the only actors in the VE which can actually change significantly over the course of their life are the room and the furniture pieces. VR-SDL uses actor graphs to model the states an object can be in (relevant to the scenario) : basically, these are an advanced form of state machines. *Figures 22* and *23* give the example actor graphs for the room and a piece of furniture.

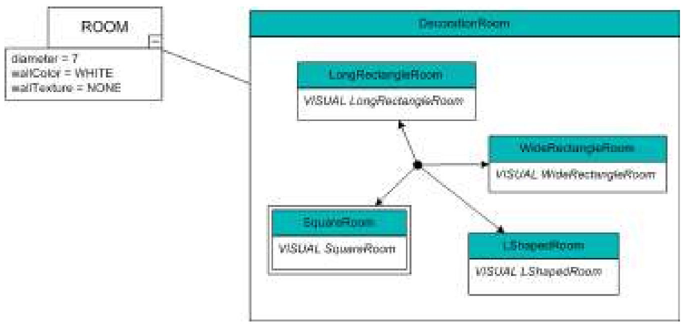


Figure 22 – Room actor graph

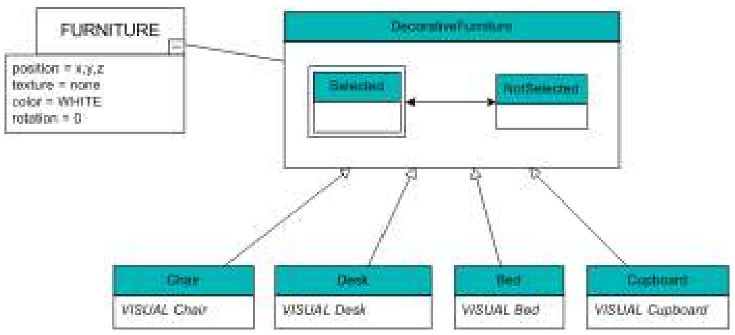


Figure 23 – Furniture actor graph

[5] A similar approach is taken in the alVRed project (section *1.4.1*)

Obviously, at this point it is impossible to understand these graphs completely, but when keeping the application in mind, trying to read them proves to be quite straightforward.

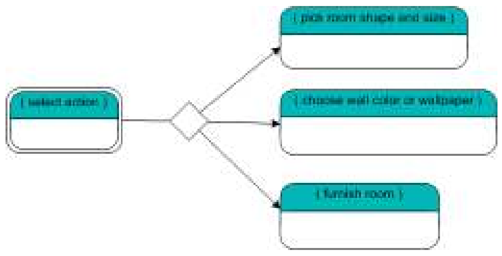
Recall that a room can have 4 possible shapes : these are exactly equal to the dual rectangles shown in the center of the room actor graph in *figure 22*. These rectangles represent the states of the room object, with the state name displayed in the top (green) box, and extra information in the bottom (white) box. From the info in the white boxes, it would appear that every state has a different visual representation (notice the keyword `VISUAL`), which is of course logical, as the states express different shapes the room can take. In addition to the shape of the room, the user can increase its size, or change its color or texture. These three parameters are precisely the ones shown in the white box to the left of the graph. They represent the room's properties.

The furniture actor graph in *figure 23* looks very similar to that of a room, with two states (*Selected* and *NotSelected*) and some properties which can be altered by the user (position and rotation, texture and color). The most striking difference when compared to the room graph are the four "states" which are attached to the bottom of the graph. The reader proficient in UML will note that the arrows connecting these boxes to the top box are identical to the UML inheritance notation – in fact, they are used for the same purpose here. The four boxes at the bottom of the diagram are not states, but child graphs. This is indeed intuitive, as chairs, beds, desks and cupboards are all furniture. The child graphs display the same states and properties as their parent graph, with the exception of having a different visual representation.

3.3.3.2. Scenario Graph

While actor graphs are essential to scenario modeling, the most important part is of course the scenario itself. In VR-SDL, scenarios are modeled using special flowcharts, containing linked events (things which happen in the VE). These events can contain pre-conditions (things that should be true for the event to be executable) and post-conditions (things that become true after execution of the event). Pre- and post-conditions are the main way in which the scenario graph is linked to the actor graphs, as conditions usually include a change in the actor graph of some actor (a state transition, a property assignment...).

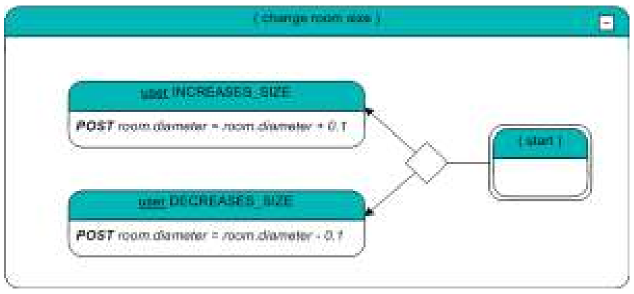
In the application description from section *3.3.2*, a clear hierarchy of the possible actions taken is discernable. At the highest level, there are three actions : change the room's shape or size, adjust the room's wall color or paper, and manipulate the furniture. *Figure 24* shows a high-level scenario, modeling this in VR-SDL. The *select action* event is the initial event, where the execution starts, the three other events are the actual actions which can be performed by the user. Note that after any of the three actions on the right are performed, execution returns to the *select action* event.



{ select action }
{ pick room shape and size }
{ choose wall color or wallpaper }
{ furnish room }

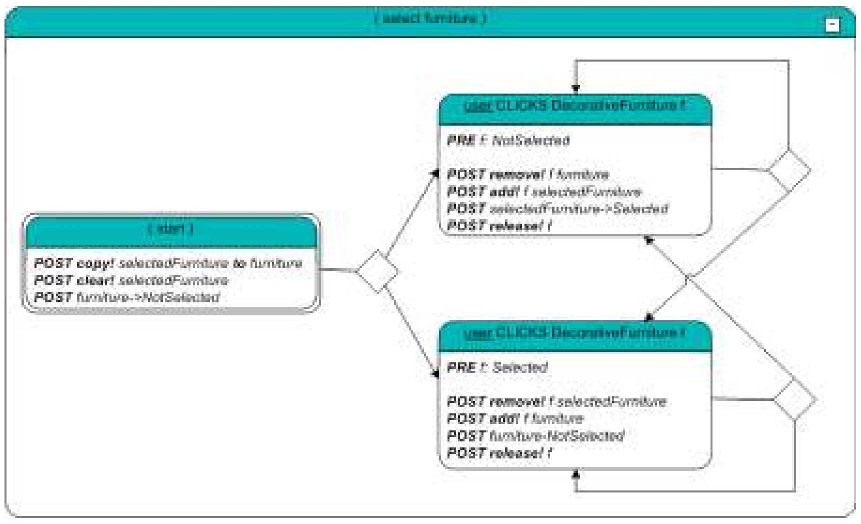
Figure 24 – Top level scenario graph

Obviously, the three events on the right of the diagram should be modeled on a lower level. The full specification of these events is quite complex, and too elaborate to show here. They can be found in appendix B. Instead of showing the full graphs, a few snapshots will be described.



{ change room size }
user INCREASES_SIZE
POST room.diameter = room.diameter + 0.1
user DECREASES_SIZE
POST room.diameter = room.diameter - 0.1
{ start }

Figure 25 – Scenario graph for modifying the size of a room



{ select furniture }
{ start }
POST copy! selectedFurniture to furniture
POST clear! selectedFurniture
POST furniture->NotSelected
user CLICKS DecorativeFurniture f
PRE f: NotSelected
POST remove! f furniture
POST add! f selectedFurniture
POST selectedFurniture->Selected
POST release! f
user CLICKS DecorativeFurniture f
PRE f: Selected
POST remove! f selectedFurniture
POST add! f furniture
POST furniture-NotSelected
POST release! f

Figure 26 – Scenario graph for selecting furniture

Figure 25 shows a part of the *pick room shape and size* event, and describes how to increase or decrease the size of a room. The user has two options : he can click either the increase or the decrease button. Clicking the increase button results in an enlargement of the room diameter by 0.1, while clicking *decrease* reduces the room diameter by 0.1. This is quite literally visible from the graph, and shows an initial coherence between the actor graphs and the scenario graph : the *diameter* property being altered is the exact same one which was displayed in the room actor graph (*figure 22*).

The graph for selecting furniture, displayed in *figure 26*, is more complicated. While the user still has only two possible actions, there are significantly more pre- and post-conditions present in this graph. Initially, regardless of which action is taken by the user, all selected events are deselected at the start. This is shown by the post-conditions in the *start* event. Basically, there are two sets, *furniture* (containing the non-selected furniture) and *selectedFurniture* (containing the selected furniture. To deselect all furniture, the elements from the *selectedFurniture* set are copied to the *furniture* set, after which the *selectedFurniture* set is cleared. Finally, all elements of the *furniture* set have their state shifted to *NotSelected*. This final post-condition demonstrates another association between the object and scenario graphs : the state change in the actor graphs of the *furniture* actors is triggered by the scenario graph.

The pre- and post-conditions present in the remaining two events of *figure 26* are very similar, except that one single object is switched from one set to another, and has its state changed. Again, this can be read quite literally from the graph.

3.4. Methodology

At this point, the reader should have a general idea about the VR-SDL approach, and some insight into the required items for modeling a scenario. An important question which remains to be answered, before going into the details of the notation, is how one should use the available constructs for modeling a scenario – in other words, which approach is followed when modeling a scenario?

3.4.1. Four Steps

The modeling process is quite intuitive, but in general, four steps can be identified : the ***description step***, the ***actor graph modeling step***, the ***actor definition step***, and the ***scenario graph modeling step***. When looking at *figure 21*, at the start of this chapter, one can see that only the last three steps are presented in the diagram. The description step is not included in the figure, since it is not formally defined, and does not yield any formal output.

3.4.1.1. Description Step

In the description step, the goal is to create an informal description of the scenario. Since this step will not be formally linked to the rest of the scenario, any technique may be used for this : a text description, a brainstorming meeting, a UML use case specification, or possibly just an idea in the head of the designer. Though this phase in the scenario modeling requires no physical output, it does have certain goals which should be attained :

- Define the number and type of the actors participating in the scenario.
- Identify the general structure of the scenario.
- Create a casual, but complete, scenario description.

3.4.1.2. Actor Graph Modeling Step

The *actor graph modeling step* consists of defining the actor graphs for the actors referenced in the scenario. The required actor graphs can be deduced from the different actor types, as specified in the description step. Actor graph modeling is described in detail in chapter *4*.

3.4.1.3. Actor Definition Step

In this phase, the actors participating in the scenario are formally defined. Actors are given a name, a type (an actor graph from the *actor graph modeling step*), and if required, they are instantiated by linking them to an actual object populating the VE. While the output of the actor definition step is not physically linked to either the actor or scenario graphs (it is simply text), it is considered to be part of the scenario graph, since the actors will be referenced in that graph. For an elaborate description of the actor definition notation, see section *5.4.1*.

3.4.1.4. Scenario Graph Modeling Step

When the actors have been formally defined, all information is present for modeling the scenario graph itself. During modeling of the graph, the iterativeness of the VR-SDL scenario modeling approach will likely become apparent, as new actors and extensions to actor graphs might be discovered. Hence, it is probable that the designer should return to the results from the previous steps, in order to improve them.

3.5. Conclusion

In this chapter, a general overview of the VR-SDL approach for modeling scenarios in VR was provided. After discussing the guidelines by which the technique was designed, the practical use of scenarios in VR was illustrated by means of an example application for room furnishing. Consequently, this example was modeled using the VR-SDL approach, to provide the reader with an overview of the notation. The chapter also presented some insights into the connections between the different concepts in the scenario modeling language. Finally, the general methodology for modeling scenarios using VR-SDL was introduced.

Overall, there are two important things to remember from this chapter :

- The VR-SDL approach contains two sub-notations :
    - **Actor graphs :** Model the evolution of the actors participating in a scenario.
    - **Scenario graphs :** Describe the scenario itself.

- Four steps are traversed when modeling a scenario :
    - **Description step :** Gives an informal description of the scenario and its actors.
    - **Actor graph modeling step :** Design of the actor graphs for the different types of actors.
    - **Actor definition step :** Formal definition of the actors in the scenario.
    - **Scenario graph modeling step :** Creation of the actual scenario graph.

Now that a general overview of VR-SDL has been sketched, it is time to go into more detail about the available concepts and notations. This will be done in two stages : in chapter *4*, the specifics of actor graphs are unfolded, and in chapter *5*, a closer look is taken at scenario graphs.

Chapter 4

Actor Graphs

4.1. Introduction

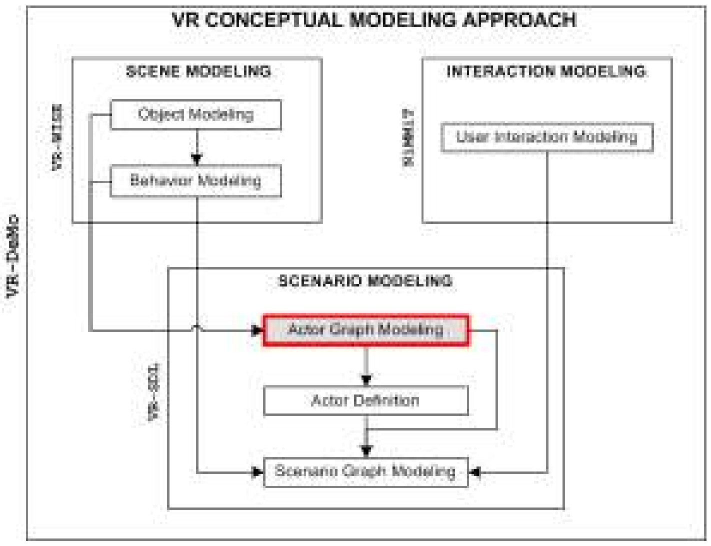


Figure 27 – Scope of this chapter : *actor graph modeling*

To model a scenario, one needs to be aware of the objects participating in it – i.e. the *actors*[6] of the scenario. These scenario actors represent the dynamic entities in the VE that can change under the impulse of the scenario, and thus, they are used to model scenario-triggered alterations to the VE. It is imperative to realize that the word "dynamic" does not imply an object can move by itself : a building or a piece of furniture can just as well be an actor in a scenario as a security guard or an enemy soldier can be. The only requirement is that the object can in some way be manipulated during the execution of the scenario : a static wall will not usually be seen as a scenario actor, but a wall which can be shot to pieces, smashed down with a hammer, or repainted, might just be.

[6] Not to be confused with behavior actors. As mentioned in chapter 3, all mentions of actors in part 2 of the thesis refer to scenario actors, unless explicitly stated otherwise.

Actor graphs are modeled from the perspective of a certain scenario – one must not make the mistake of trying to model the real world. For instance, while real walls are definitely repaintable, if you are modeling a war simulation, this is irrelevant – all that you care about (and thus, all that should be modeled) is that they can be shot to pieces. On the other hand, when modeling the room furnishing application discussed in the previous chapter, it does not matter that real walls can be shot to pieces, since this will never happen in your application. In this case, only the recoloring of walls will be modeled.

In VR-SDL, the actors in a scenario are modeled using actor graphs. This chapter is dedicated to the explanation of the actor graph notation (*figure 27*). It opens with a brief explanation of the actor graph modeling approach in general, after which the different components are described in detail. This is done as comprehensively as possible, so the interested reader is provided with an explanation of the reasoning behind the introduction of certain constructs, and examples to clarify them where feasible. Readers who are not looking for so much detail may limit themselves to sections *4.2* and *4.12*, to obtain a general idea of the approach for modeling actor graphs.

4.2. Approach

As a simple example of an actor graph, consider a castle guard with two states, *Friendly* and *Hostile*, depicted in *figure 28*. Depending on the user's actions (as modeled in the scenario graph), the guard could be in either of these two states.

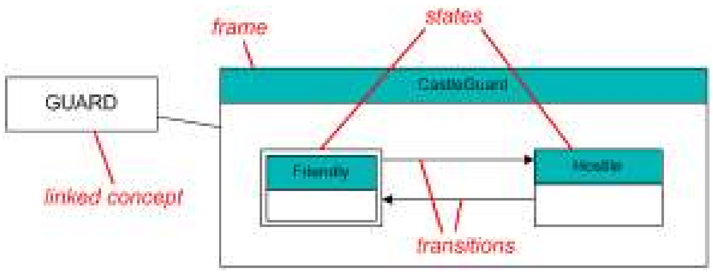


Figure 28 – Actor graph example : castle guard[7]

Actor graphs consist of four important elements : a *frame*, a *linked concept*, *states*, and *transitions*.

[7] The red names and lines are not a part of the VR-SDL notation

An actor graph is always encased in a dual or triple box, called its ***frame***, which contains the graph's name in its top box (in *figure 28*, this is *CastleGuard*), and the states and transitions in the bottom one. Additionally, the frame may contain property definitions and subobject initializations (in an optional third box, below the previous two), and the linked concept is graphically connected to it. Frames are elaborated upon in section *4.3*.

Each actor graph has exactly one ***concept*** linked to it. This is a lower-level concept, modeled using the VR-WISE approach (as described in section *2.3*). It represents the graphical composition of the object, and possibly some concept properties. Both simple and complex concepts are available for linking to actor graphs. For more information, refer to section *4.4*.

The main building blocks of an actor graph are its ***states***. States are represented by a double rectangle, the top box of which contains the state's name, while the bottom may contain qualifications, behaviors, visuals, or nested subgraphs. States and their contents are explained in sections *4.5*.

States are connected by means of ***transitions***. A transition connects two or more states, and indicates a possible state change from the start state to the end state. Transitions may contain a linked behavior or a trigger expression. However, most transitions do not contain a trigger, since the impulse for a state change will usually be given on the scenario level (and thus, it will be modeled in the scenario graphs). A more detailed description of transitions can be found in section *4.10*.

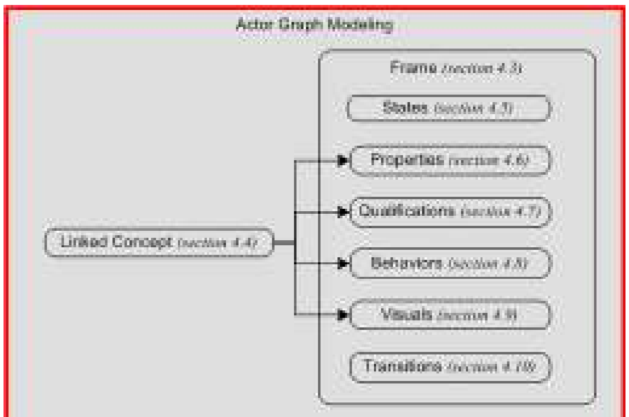


Figure 29 – Components of an actor graph

Figure 29 graphically illustrates the different components of an actor graph, and the sections they are described in. The arrows signify that *properties*, *qualifications*, *behaviors* and *visuals* may depend on the *linked concept*. Note that this graph may be substituted for the grey box in *figure 27*. The arrow in *figure 27*, going from *object modeling* to *actor graph modeling*, would then be connected to the *linked concept* in *figure 29* (since the linked concept is a concept modeled on the VR-WISE level). The arrow in *figure 27* from *behavior modeling* to *actor graph modeling* would be connected to *behaviors* in *figure 29*, since these are dependent on the some lower-level VR-WISE behavior. This is explained in detail in the corresponding sections.

4.3. Frame

Each actor graph is encased in a top-level box, called the ***frame*** of the graph (*figure 30*).

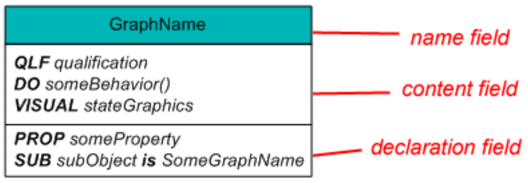


Figure 30 – Frame notation

The frame consists of two mandatory boxes, the ***name field*** and the ***content field***, and an optional third box, called the ***declaration field***. Additionally, the frame is always connected to the actor graph's linked concept.

- The **name field**[8] contains the name of the actor graph. It is this name which is used to refer to the actor graph, for example, to specify the *role* of a scenario actor (section *5.4.1*).
- At the very least, the frame's **content field** contains the actual states and transitions making up the graph. Additionally, the frame's content field may include everything that can be added to a regular state's content field : qualifications, behaviors, and visuals. These constructs are described in detail throughout the remainder of this chapter.
- The **declaration field** is optional, and is only present if the actor graph contains *actor graph properties* (section *4.6.3*) or if the *linked concept* is complex (section *4.4.3*).

[8] The name field is colored turquoise in the remainder of this thesis. This coloring is not required, it merely clarifies the notation.

4.4. Linked Concept

An actor graph's ***linked concept*** determines which kinds of objects from the VE are allowed to function as the type of actor defined by the actor graph. Since the linked concept is simply a concept modeled on the lower level (section *2.3*), it forms the connection between the actor graph modeling phase, and the lower level object and behavior modeling phases.

4.4.1. Notation

Linked concepts are represented by a rectangle (*figure 31*), in analogy with the concept notation from VR-WISE object modeling. Per convention, the concept name is written in capitals. The concept is linked to the actor graph's frame by a single line.

Figure 31 – Linked concept notation

4.4.2. Actor Graphs as Concept Profiles

As described in chapter *2*, concepts are used to model the types of objects populating the world, and are instantiated to create the actual entities in the VE. For scenario modeling, however, an additional step is required. Suppose that for example, a concept *GUARD*[9] has been defined. It is not at all unimaginable that several different kinds of guards are required in a scenario : patrolling guards, cowardly guards who get scared instead of fighting, bodyguards who protect a person instead of patrolling along a route, ... All these different types of guards define different profiles for the *GUARD* objects in a scenario.

Modeling these profiles is exactly what actor graphs are used for. In this particular example, three actor graphs would be created with a linked *GUARD* concept : a *PatrolGuard* graph, a *ScaredGuard* graph, and an *Bodyguard* graph. Each of these graphs would define different states for their particular type of guard, thus creating a distinction between them. This distinction might be quite radical : it is obvious that a patrolling guard may behave entirely different during the course of the

[9] From now on, concept names will always be denoted in capital letters, to distinguish them from state names.

scenario than a bodyguard. At the same time, the fact that each of these guards is linked to the same lower level concept causes them to display certain similarities, i.e. the graphical representation of the guards, their properties, and possibly linked behaviors (more on the latter in section *4.8.4*).

The objective of actor graphs is to define the different types of actors present in the scenario. Hence, an actor graph is not an instance, and therefore, it should be instantiated before being usable within a scenario. Instantiating a scenario actor is done by linking an instance of its linked concept from the VE to the actor graph. For instance, if there are four *GUARD* objects in the VE, called *guard1*, *guard2*, *guard3*, and *guard4*, each of these could be linked to any of the guard actor graphs, thus creating a usable actor for the scenario. A full description on defining actors is given in section *5.4.1*.

4.4.3. Complex Concepts

4.4.3.1. Referencing Subobjects in Actor Graphs

Naturally, an actor graph's linked concept could be a complex concept. If this is the case, it should be possible to reference the subobjects of the concept, so their properties, behaviors etc. can be used within the actor graph. Below, the available notations for referencing subobjects in an actor graph are shown :

```
<subobject-ref> ::= <role-name>
                  | <concept-name>
                  | <actor-graph-name>
                  | <subobject-ref>.<subobject-ref>
```

If a concept contains multiple subobjects instantiated from the same concept (such as a table with four legs), each of them is given a unique role name (section *2.3.2.2*). If there is only one subobject instantiated from a certain concept, a role name is facultative. However, in such a case, the concept name is unique. Thus, the subobjects of a linked concept are uniquely referable by using either their *role name* or *concept name*. For continuity, subobjects are always given a role name throughout this thesis.

In the event of multiple subobjects instantiated from the same concept, the concept name may still be used for referencing the subobjects : it will simply return the set of all subobjects instantiated from the correct concept.

Since subobjects should be linked to an actor graph themselves (as explained in section *4.4.3.2*), *actor graph names* can be used to reference them as well. The interpretation is completely analogous to using the concept name : referencing subobjects through their actor graph names, returns the set of all subobjects linked to that actor graph. Due to the similarity to referencing through concept names, this notation will be largely disregarded in the rest of this thesis.

To provide an overview of the available subobjects, with their concepts and names, the linked concept notation is extended with a drop-down box, similar to the one in the object modeling phase. As an example, *figure 32* illustrates a guard, modeled as a union of a head, a torso, two arms, two legs and a sword.

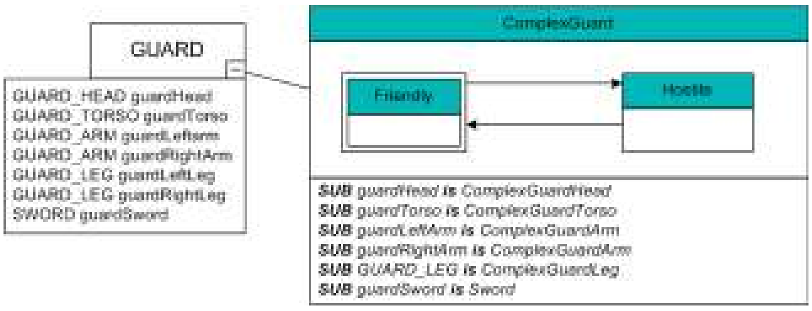


Figure 32 – Complex object example

The items in the *GUARD* concept's drop-down box are written in a `<concept> <role>` notation, meaning the names used to reference the subobjects within the actor graph are *guardHead*, *guardTorso*, *etc*. Alternatively, the concept names *GUARD_HEAD*, *GUARD_TORSO*... may be used. Note that using *GUARD_ARM* or *GUARD_LEG* returns a set with two elements.

If a subobject in turn has subobjects itself, these can be referenced by using a dot-operator. For example, suppose that an arm has subobjects *hand*, *lowerArm*, and *upperArm*, then the guard's left hand is referable through the expression *guardLeftArm.hand*.

It is important to realize that the linked concept's drop-down box exclusively contains concept-specific information, in other words, things which are not part of the actor graph modeling, but are modeled on the lower level. Later in this chapter, other items which can be displayed in this drop-down box are mentioned. Since the drop-down box only contains an overview of existing elements of the *GUARD* concept, and no new scenario modeling elements are ever displayed in it, the notation of its contents are not formalized. It may for example be dependent on the tool used to create the scenario diagrams.

4.4.3.2. Linking Subobjects to Actor Graphs

Simply mentioning the existence of subobjects is not enough : if an actor graph's linked concept is complex, all of its subobjects should be assigned to an actor graph as well. This is done by modeling the required actor graphs separately, and adding statements linking subobjects to these actor graphs to the initial graph's declaration field. These statements are of the form

```
<subobject-link> ::= SUB <role-name> is <actor-graph-name>
                   | SUB <concept-name> is <actor-graph-name>
```

The first type of statement links one particular subobject to an actor graph, while the second type links all subobjects of a certain concept to the same actor graph. This second notation can cause a drastic reduction in the number of statements needed if a concept contains many subobjects of the same type (for example, a centipede with dozens of legs).

Figure 32 illustrates both notations. Note that the first form is used for linking the arms to the *ComplexGuardArm* actor graph, while the second form is used for the legs. As you can see, this means that two statements are needed for the arms, while a single one suffices for the legs.

4.4.3.3. Unconnected Complex Objects

Since unconnected complex objects (section *2.3.2.3*) behave in the exact same way as connected ones, they are modeled similarly. For example, consider a platoon of guards, displaying coordinated behavior (such as using intricate strategies to sneak up on the enemy, or regrouping when one of the platoon's members has been killed). While each guard is clearly a stand-alone object, modeling them all separately is insufficient. Designing the platoon as an unconnected complex object is the obvious solution to this problem.

Using complex linked concepts leads to numerous other issues, which will be described in the appropriate sections throughout the remainder of this chapter. Keep in mind that all of these descriptions hold for both connected and unconnected complex objects.

4.5. States

4.5.1. Notation and Contents

A state in an actor graph represents a certain status of the object during the execution of the scenario. The notation of a state is very similar to a frame (*figure 30*). States are drawn as dual rectangles, with a name and a content field. Unlike frames, states never have a declaration field.

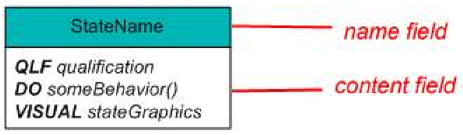


Figure 33 – State notation

The ***name field*** contains the state's name, per convention written in the same style as Java classes (e.g. *StateName*). In addition to being simple to interpret for non-programmers, this notation provides the advantage of being intuitive for programmers with notions of UML. This is due to the analogy they will undoubtedly make between states (the main building blocks of actor graphs) and classes (the main building blocks of UML *class diagrams*). Note that the same naming convention is used for the name of the graph, entered in the frame's name field.

The state's ***content field*** may contain qualifications (section *4.7*), behaviors (section *4.8*), visuals (section *4.9*), and nested subgraphs. The first three items use a textual notation, while subgraphs simply consist of a number of linked states – in other words, just like a regular actor graph, except that they do not contain a frame or linked concept.

4.5.2. Start State

Obviously, an actor graph should always have a ***start state***. This is the state in which the object initially resides. Start states are drawn like regular states, but with a double edge (*figure 34*).

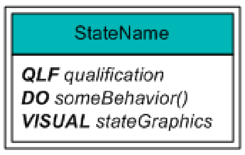


Figure 34 – Start state notation

Aside from the actor graph itself, every subgraph should have exactly one start state as well.

4.6. Properties

An object can have certain data associated with it, called ***properties***. Properties can either be strings or scalars (***free properties***), or values from a predefined set (***enumeration properties***). Properties are not statically typed : a free property may be assigned a string value at some point during its existence, and a scalar later on. On the other hand, enumeration properties can only take on values from the specified domain.

4.6.1. Scope

Properties can be divided into two groups based on their scope : they are either assigned to an entire concept (***concept properties***) or to a single actor graph (***actor graph properties***). Concept properties are modeled on the lower level, during the object modeling phase, while actor graph properties are defined within the actor graph itself. However, within the graph, both can be utilized in the same way.

Consider the guard mentioned in section *4.2*. A guard (an instance of the *GUARD* concept) could have several general characteristics, such as his *strength*, *stamina*, or *courage*. These are concept properties of the *GUARD* concept. However, suppose there is a scenario requiring two different types of guard : a patrolling guard, and a bodyguard. Both are clearly guards, so they have the properties described above, but they act differently – for example, in case of turmoil, the patrolling guard would go investigate, while the bodyguard would try to secure his client. This means that both types of guards have a different actor graph.

It is not at all unthinkable that one would like to include properties in one of the actor graphs which are irrelevant to the other. For example, the bodyguard could have a *loyalty* property, determining how dedicated he is to protecting his client, while the patrolling guard could have a *curiosity* property, indicating the probability that he will get distracted from his regular patrol route. Both properties would obviously be useless to the other type of guard. Hence, *loyalty* and *curiosity* are actor graph properties of their corresponding actor graphs.

Figures 35 and *36* show two actor graphs modeling these guards. The notations are described in sections *4.6.2* and *4.6.3*.

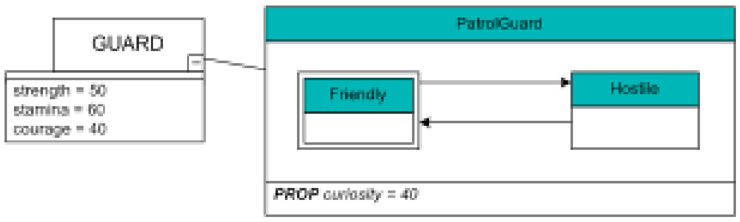


Figure 35 – Patrol guard actor graph

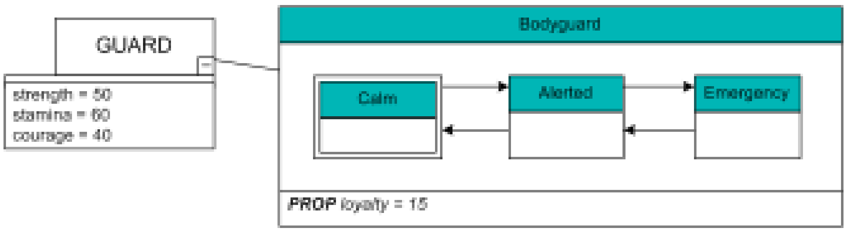


Figure 36 – Bodyguard actor graph

4.6.2. Referencing Concept Properties in Actor Graphs

4.6.2.1. Simple Concepts

Since concept properties are defined on the lower level, the designer can get an overview of a concept's available properties in the linked concept's drop-down box (as displayed in *figures 35* and *36*), in the same way as subobjects (also modeled on the lower level) are visible there. Remember that concept properties may have an initial value assigned to them.

Since concept properties all fall within the same scope, they have a unique name. Thus, for properties of the top-level concept, this name can be used to reference them within the actor graph.

4.6.2.2. Complex Concepts

For complex concepts, things are slightly more complicated. Like simple concepts, complex concepts can contain concept properties. The properties of the complex concept itself are no problem : since they are uniquely named within the scope of the top-level concept, they can be referenced in the exact same way as properties belonging to a simple concept. However, the subobjects of a complex concept might contain properties of their own, and the scope of these properties is limited to the concept they belong to. Therefore, using only their name to reference these properties might give rise to naming conflicts. This is solved by adding a subobject reference (as described in section *4.4.3.1*) in front of the property name, with a dot-operator :

```
<complex-prop-ref> ::= <subobject-ref>.<property-name>
```

The dot-operator will be the standard VR-SDL notation for accessing a certain segment of an item.

Consider the complex guard from *figure 32*, for example, and suppose the *GUARD_ARM* concept contains a property *length*. Simply using the name *length* within the *ComplexGuard* actor graph is not possible, since there are at least two different *length* properties within the linked *GUARD* concept (one for each arm). Additionally, other subconcepts may also contain a *length* property (the *SWORD* or *GUARD_LEG* concepts, for example). However, since property names are unique within their own concept, the combination of an object reference and a property name provides unique identifiers for the properties – in this case, *leftArm.length* and *rightArm.length* (and possibly, *leftLeg.length*, *rightLeg.length* and *guardSword.length*).

4.6.3. Defining Actor Graph Properties

While concept properties are existing properties from the lower level modeling, actor graph properties should be defined within the actor graph itself. This is achieved by adding actor graph property definition statements, labeled with the keyword `PROP`, to the graph's declaration field. A newly defined property should be given a name, a possible initial value, and in the case of an enum property, the possible values should be specified as well.

A property definition statement is written as

```
<free-prop-definition> ::= PROP <property-name>
                         | PROP <property-name> = <value>
```

In case of an enum property, this is extended to

```
<enum-prop-definition> ::= PROP <property-name> {<enum-list>}
                         | PROP <property-name> {<enum-list>} = <value>

<enum-list> ::= <enum-value>
              | <enum-value>, <enum-list>
```

Note that the initial value assignment is optional in both cases.

Once an actor graph property has been defined, its name is available to reference it within the entire actor graph. However, this also means none of the property names should conflict with any of the linked concept's properties, which are referenced in the same way. For example, in *figures 35* and *36*, it is impossible to define an actor graph property called *strength*, since this name is already taken by a concept property.

4.7. Qualifications

When modeling the possible evolution of an actor in a scenario through a state machine, one would obviously like to add content to these states. After all, saying that a guard can be friendly or hostile has very little meaning without defining what it means to be friendly or hostile – and particularly, what separates one state from another one. To specify this, states can be filled with visuals, behaviors, or qualifications. The latter is the subject of this section.

Qualifications are a way to specify requirements for a state. Informally, some example qualifications could be "while in the friendly state, a guard's curiosity is equal to 30", "a hostile guard has a courage between 25 and 50", or "when a guard is hostile, his face has an angry look". Two types of qualifications are distinguished : property qualifications, and state qualifications.

4.7.1. Property Qualifications

As the name suggests, ***property qualifications*** define constraints on a certain property. More specifically, they assign a value to a property. Remember though, that enumeration properties can only take values from a pre-defined domain.

Assigning a new value to a property can be done in two ways :

- Fixed value assignment. Quite simply, a fixed value is assigned to the property. For strings and enumeration properties, the value is always a simple static value, but for scalars, it may be the result of an expression based on the value of other properties (or even recursively dependent on the property being assigned). Notation :

```
<prop-single-qlf> ::= QLF <prop-ref> = <value>
```

- For scalar assignments, random value assignment is also available. In this case, min and max values are defined, and the property is assigned a random value from within this interval. The precision of the final result (and thus, the amount of variation possible) is equal to the most precise argument. In other words, a lower boundary of 30 and an upper boundary of 50 will cause an assignment of a random integer from the interval [30,50], but if the lower boundary is defined as 30.0, a floating point value with a decimal place of 1 (thus, a value from the set {30.0, 30.1, ..., 50.0}) will be selected. Notation :

```
<prop-random-qlf> ::= QLF <prop-ref> = [ <num-value>,
                            <num-value> ]
```

Figure 37 shows the use of both assignment types in practice. When entering the *Friendly* state (since *Friendly* is the start state, this includes object initialization), the value of the *curiosity* property is decreased by 20, and a random value between 15 and 50 is assigned to *courage*. When the guard goes hostile, his *curiosity* is increased by 20, and all other properties remain unchanged. Note that, when the guard changes back from *Hostile* into *Friendly*, *courage* is once again assigned a random value. Hence, if the object keeps changing back and forth, *curiosity* will only ever take two values (20 and 40), while *courage* will keep changing pseudo-randomly.

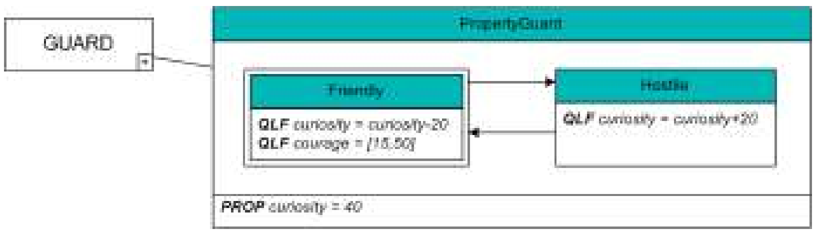


Figure 37 – Patrol guard, extended with property qualifications

4.7.2. State Qualifications

The second type of qualifications, ***state qualifications***, are reserved for actor graphs with complex linked concepts. When a certain state, called *someState*, contains a state qualification, this indicates that some subobject is in a particular state for as long as its parent object remains in *someState*.

4.7.2.1. Single State Qualifications

The most straightforward type of state qualifications, called ***single state qualifications***, indicates that a subobject is in exactly one state. Single state qualifications are written as follows :

```
<single-state-qlf> ::= QLF <subobject-ref> : <state-ref>
```

The subobject reference is identical as specified before : subobjects can be referenced by their role name, concept name, or their actor graph name. The first approach returns a single object, while the latter two may return a set. Referencing a set of subobjects in a state qualification signifies that all objects in the set are in that state.

Like in section *4.4.3.2*, the set notation is introduced to greatly reduce the number of statements needed in case of a large number of subobjects with similar behavior (for example, a number of small lights in a christmas tree). It should be noted that the single object notation has priority over the set notation : if a state contains a single state qualification about a particular subobject (denoted by its role name) and another one about the concept this subobject belongs to, it is the former qualification which holds for the object.

Figure 38 demonstrates single state qualification notations. It displays an office with a door and two windows, a left one and a right one. Each of these have actor graphs with two states, *Normal* and *Burning* (the actor graph for both windows is the same). During the course of the scenario, either the left or the right window can catch on fire.

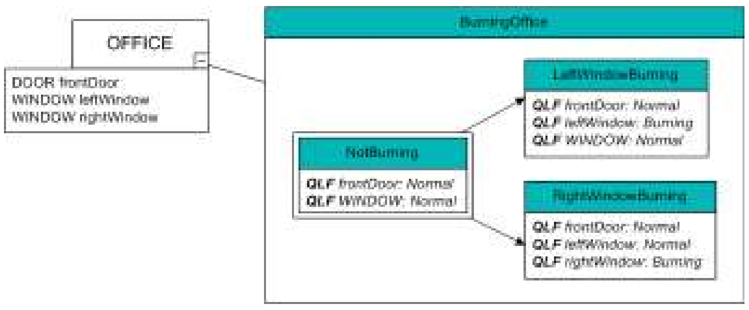


Figure 38 – Burning office displaying state qualifications

Note that, for demonstration purposes, the approach taken for modeling the *LeftWindowBurning* state is different from that used for the *RightWindowBurning* state. The latter uses the role notation for each qualification, while the former uses a concept notation to state that all subobjects of concept *WINDOW* are normal in the *LeftWindowBurning* state, except *leftWindow*, which is burning. This is an example of the role notation getting priority over a set notation. This modeling approach may seem silly, and in this particular case, it arguably is. However, the notation's usefulness quickly becomes apparent when considering an *OFFICE* concept with – say – ten windows instead of two.

4.7.2.2. Multiple State Qualification

The single state notation has a problem : it may lead to an explosion in the number of states needed. To understand why, let us revisit the burning office example.

Suppose that the office's door and windows have an additional state, *Doused*, and that the scenario is a game where the player is a fireman, dropped in a room where every object is burning, and tasked with extinguishing the burning office, one object at a time. On a first hunch, one might attempt to model the office actor graph for this scenario as shown in *figure 39*.

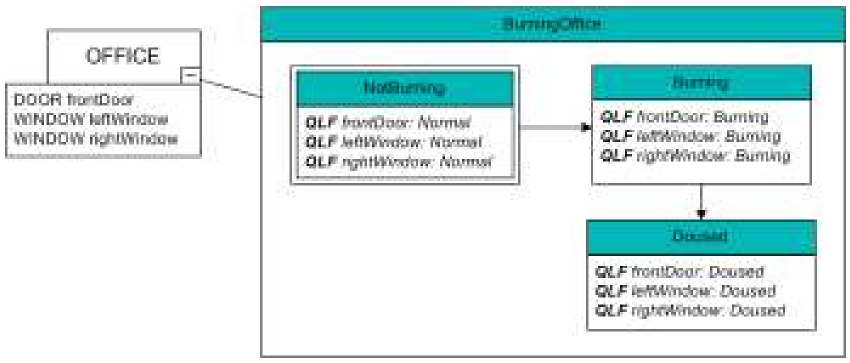


Figure 39 – Badly modeled burning office actor graph[10]

However, this office model presents a problem : during execution of the scenario, player interaction might lead to the office object being "between states". For example, it is possible that the player puts out the front door and one of the windows, but not the other window. In this case, the burning office is neither in the *Burning*, nor the *Doused* state.

It is entirely unpractical to include a new state for each of the possibilities. In this case, even though there are only 3 subobjects, each with a mere 2 relevant states (*Burning* and *Doused*, since the player can not revert the objects to the *Normal* state by interacting with them), this would lead to 6 extra states in the actor graph : one for each possible combination of states for all the subobjects (there are in fact 8 combinations, but the cases where the subobjects are all burning or doused are already present). *Figure 40* illustrates this new situation. Note that using the concept notation for the state qualifications here instead of the role names is not allowed : the whole point is that every subobject's state should be separately changeable, if the concept notation would be used, both windows would always be in the same state.

[10] This graph exclusively uses the role notation for single state conditions. This is purposely done, for clarity, and is unrelated to the graph being badly modeled.

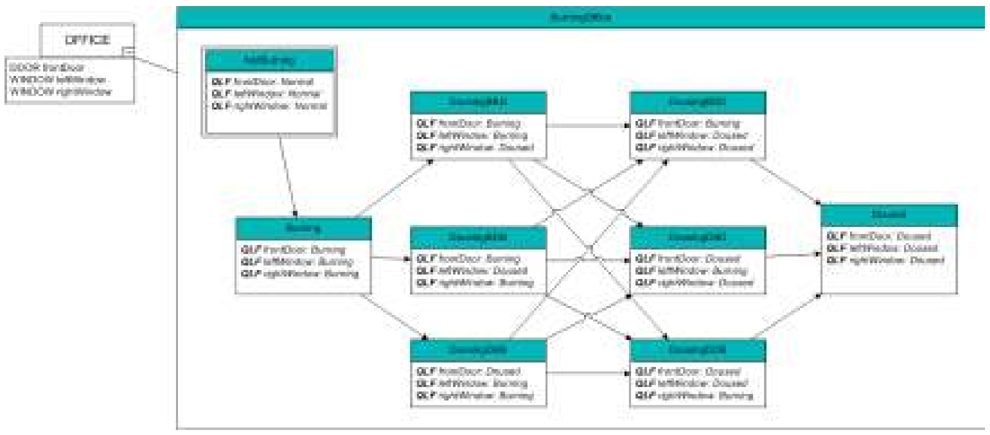


Figure 40 – Example of the cluttering caused by adding a new state for each possible combination of subobject states.

Now imagine that the office would have ten windows, instead of two, for a total of 11 subobjects; and that doors and windows and door can not only be doused by the player, but also smashed, adding a third possible state to each subobject. In this case, the amount of required states would equal 3^{11} = 177147 states. Clearly, this is unfeasible to model using the current approach.

There are two possible 'solutions' : either accept *figure 39* as a correct graph after all, and allow this 'hovering between states'. This is a pretty serious case of non-determinism, however, as objects could be in states which don't really exist. In fact, this is not a solution at all : the problem is merely being swept under the carpet.

The second option, which is adopted in VR-SDL, is to introduce a new type of state qualification, which allows for a subobject to be in a 'set of states'. This means that the object it is in actuality in one of the states from the set, based on some event from the scenario. This still implies non-determinism, but far more controlled. Not only is it now known in which state the object is, but also which set of states the subobjects may be in. In fact, in combination with the executed scenario, each object's state will be known.

This new type of qualification is called a ***multiple state qualification***, and it is denoted as follows (note that both a single or multiple objects may be referenced again) :

```
<multiple-state-qlf> ::= QLF <subobject-ref> : { <state-list> }

<state-list> ::= <state-ref>
               | <state-ref>, <state-list>
```

Figure 41 shows the final, correct actor graph for the office, modeled using multiple state qualifications. Observe that the cluttering is drastically reduced when compared to *figure 40*.

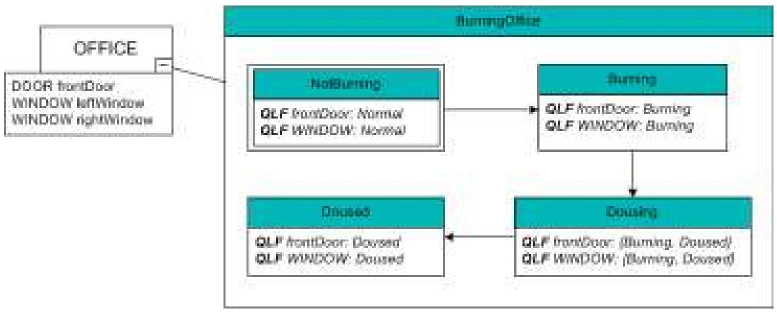


Figure 41 – Correct remodeling of the office graph, using multiple state qualifications

4.8. Behaviors

Consider the aforementioned patrolling guard : in *figure 37*, a possible actor graph is given for this guard, but in neither of the two states, any indication is given that the guard is walking along a patrol route. Additionally, one would expect a hostile guard to take action against his enemies, but this is not present in the graph either.

To be able to model situations like those described above, ***behaviors*** are introduced to the actor graph modeling. Like properties, they are a way to further detail the graph's states, and to distinguish them from one another.

This section is divided into four subsections. Section *4.8.1* provides an introduction to behaviors in actor graphs, detailing the different behavior types and illustrating them by means of the guard example. Afterwards, a more in-depth description of the notations of the various behaviors is given in section *4.8.2*. Consequently, section *4.8.3* provides more insight into the way behaviors are executed, and the related modeling issues. Finally, section *4.8.4* clarifies the link with the lower level behavior modeling phase in VR-WISE (as described in *2.4*)

4.8.1. Behavior Types

Initially, let us assume that the designer is provided with a selection of behaviors (modeled on the lower level) which can be performed by the concept linked to the actor graph. Like all concept-related items modeled on a lower level than the scenario modeling, an overview of the available behaviors is shown in the concept's drop-down box.

Within an actor graph, behaviors are generally labeled with a `DO` keyword[11], and are given a name for referencing purposes. In general, the notation is

```
DO <behavName> = <specification>(<parameters>)
```

where `<behavName>` is a random string, `<specification>` is the name of the lower level behavior, and `<parameters>` is an optional list of parameters (more explanation on this in section *4.8.4*).

Under the assumption above, *figure 42* displays an example of behaviors in actor graphs, by providing an extended graph for the patrolling guard described earlier in this chapter. This example contains several different types of behavior.

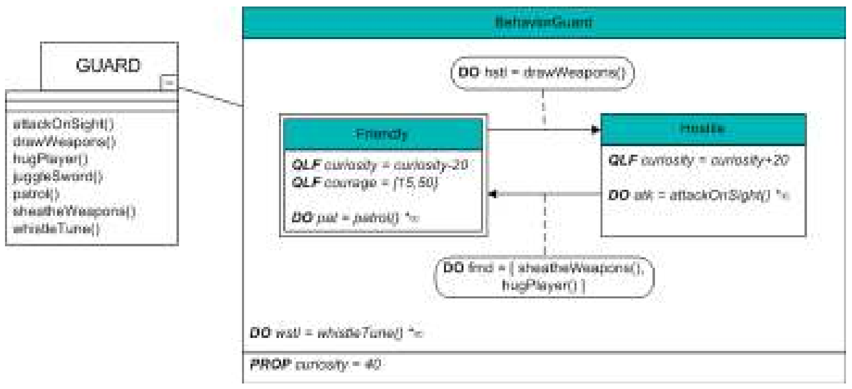


Figure 42 – Patrolling guard actor graph, extended with behaviors

First and foremost, the *Friendly* and *Hostile* states each contain a ***state behavior***. In general, state behaviors – such as *pat* and *atk* – are behaviors which are executed while the object is in the state that contains them. In this particular case, the *pat* behavior implies that the guard is patrolling along a predefined route while he is in the *Friendly* state, while the *atk* behavior signifies that in the *Hostile* state, he attacks hostile targets on sight.

Obviously, it is possible that one would want to attach a behavior which remains constant over all states. Suppose, for example, that the guard from *figure 42* is very upbeat, and continuously whistles a tune, regardless of whether there are enemies nearby. This is modeled by means of a ***global behavior***, which is nothing more than a state behavior of the graph's encasing frame – and thus, it is always

[11] There are also CONT behaviors, these are explained in section *4.8.3.1*.

executed, regardless of which state the object is in. The *wstl* behavior in *figure 42* is an example global behavior.

Not only the states can contain behaviors, but also the transitions between states : these behaviors are called ***transition behaviors***. They are written in a rounded rectangle, which is connected to the appropriate transition by a dashed line. Unsurprisingly, transition behaviors are executed when the object switches from the transition's start state to its end state. In *figure 42*, the guard will draw his weapons when he turns hostile, and hug the player after sheathing his weapons when becoming friendly.

This last behavior (called *frnd* in the graph) displays another interesting quality : it is composed out of two independent behaviors. Such behaviors are called ***composite behaviors***. Composite behaviors are simply sequences of a number of behaviors, executed serially : the next behavior in the sequence is only started when the previous one has completed. Composite behaviors only have a single name, and both state and transition behaviors can be composite.

4.8.2. Notation of Behaviors

After presenting an overview of the different types of behaviors usable in actor graphs, it is time to revisit the notation in more detail. Primarily, the notion of an execution counter is introduced, and after that, the notation for behaviors is described formally.

4.8.2.1. Execution Counter

State behaviors are executed at least once, but they may also be looped several times – or possibly even infinitely. The amount of times they are executed can be specified by the designer : a behavior is followed by an optional ***execution counter***, indicating how many times it is looped. An execution counter is written as **x*, where *x* is an integer or ∞.

For example, if the patrol behavior in *figure 42* were written as '*patrol() *1*', the guard would run his patrol route once, and then stand idle. If '*patrol() *3*' were used, he would run it three times, and so on. It is also possible to create a behavior which is looped infinitely : this is what the execution counter *∞ is used for. The default amount of times a behavior is executed is one, so it is allowed not to specify an execution counter: in this case, the behavior is simply executed once.

For composite behaviors, it is possible to specify an execution counter for the entire behavior sequence, and an additional execution counter for each separate behavior within the sequence. This implies that it is possible to write a behavior such as '*[danceMove() *2, clapHands() *1] *∞*', which would result in an infinite loop of the object performing 2 consecutive dance moves, and clapping hands after that. Note also that in this particular case, the execution counter after *clapHands()* may be omitted.

It may at first glance appear as though an execution counter is too limiting, and additional constructs are needed : granted, behaviors with much more elaborate timing are definitely imaginable. Consider, for example, a guard with ten different idle animations (scratching his head, looking around, yawning, ...), each of which should be displayed pseudo-randomly, and within randomly timed intervals of each other. Clearly, this is impossible to model using nothing but an execution counter and complex behaviors.

However, one must not forgot that a very complete, well thought through behavior modeling technique is already present in VR-WISE (section *2.4*). There is no need for trying to reinvent the wheel by formulating an overload of extra behavior modeling constructs within the actor graph notation : at best, this would lead to a trimmed down version of a more complete, already existing approach, and in the worst case scenario, it would cause inconsistencies between the behavior modeling on the two different levels, and severely complicate the notation by introducing an abundance of needless constructs. Thus, rather than ending up with an inferior, trimmed down version of the present behavior definition language, the existing one is used to define the behaviors on a lower level, and these behaviors are used within the actor graphs. This approach offers a great deal of flexibility, while still keeping the notation compact and intuitive.

4.8.2.2. Full Notation

In section *4.8.1*, a rough overview of the behavior notation was given. It is now time for a more complete explanation.

The standard behavior notation uses a `DO` keyword, a reference name, and a behavior definition part. At the very least, this definition part consists of the name of a lower level behavior, followed by closed parentheses. Possibly, a list of parameters is added inside the parentheses (consisting exclusively of actors participating in the behavior, as explained in section *4.8.4*) and an execution counter is attached to the back. Finally, in the case of a composite behavior, multiple of these behavior definitions are separated by commas, and the whole is enclosed in brackets.

Formally, this translates to :

```
<do-behavior> ::= DO <behav-name> = <behav-def-ctr>
                | DO <behav-name> = [ <behav-list> ]

<behav-list> ::= <behav-def-ctr>
               | <behav-def-ctr>, <behav-list>

<behav-def-ctr> ::= <behav-definition>
                  | <behav-definition> <execution-counter>

<behav-definition> ::= <behav-specification> ()
                     | <behav-specification> ( <actor-list> )
```

Aside from regular behaviors, there are also *continued behaviors*. These are used to indicate the continued execution of a behavior initiated in a different state. Their notation is very simple :

```
<cont-behavior> ::= CONT <behav-ref>
```

The philosophy behind continued behaviors is explained in detail in section *4.8.3.1*.

A final concern involves the scope of the reference name given to a behavior. It is defined to be equal to the state the behavior appears in, plus its outgoing transitions. In other words : A state and its outgoing transitions may not contain the same behavior name more than once, but names may be reused between states.

This limited scope implies the need for a scope delimiter, so the designer has the possibility of referencing a behavior outside of a state. The scope delimiter used is the double colon operator (noted as '::'), which programmers will be familiar with from languages such as C++. Hence, the full specification of a behavior reference is as follows :

```
<behav-ref> ::= <behav-name>
              | <state-ref> :: <behav-name>
```

In summary, consider *figure 42* once more. Since the *pat* and *wstl* behaviors fall within the same scope, their names are required to be different. However, should one be so inclined, these names may be reused instead of *atk* and *frnd*, respectively : both of these behavior fall within an entirely different scope. To reference the *pat* behavior within the *Hostile* state, one would write *Friendly::pat*.

4.8.3. Execution of Behaviors

Now that the details of the notation are known, some additional modeling issues, related to the way behaviors are executed, will be explored. The bulk of this section concerns state behaviors, but a short overview for transition behaviors is added as well.

4.8.3.1. Modeling State Behaviors

It is possible for multiple behaviors to be defined within a state. In this case, they are all executed simultaneously. For example, suppose the guard from *figure 42* should juggle his sword while walking his patrol route. This could be specified by adding a behavior '*DO jgl = juggleSword() *∞*' to the *Friendly* state. Now, whenever the guard is in the *Friendly* state, three behaviors are being executed : *pat*, *jgl*, and *wstl* – since the latter is a global behavior, it is always executing! Note that, when modeling states with multiple behaviors, care should be taken that no conflicting behaviors are added to the same state (like a patrol and an attack behavior). This, however, is the responsibility of the designer.

A slight adjustment of the situation described above yields *figure 43*. In this case, instead of patrolling and juggling his sword indefinitely, the guard walks along his patrol route twice, while performing the juggle routine three times.

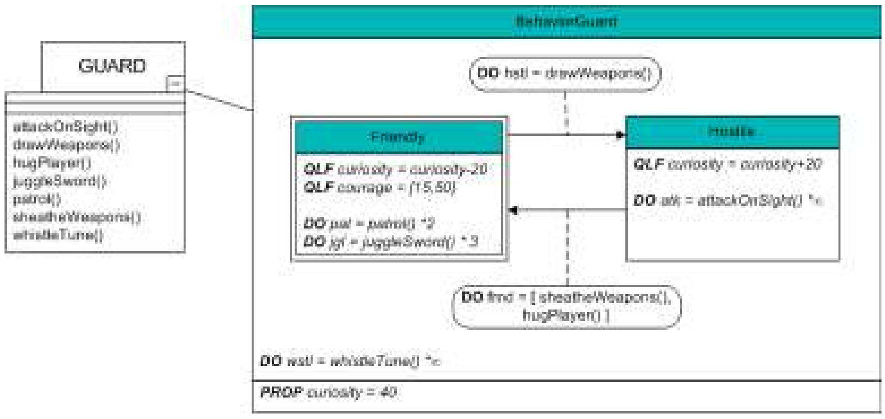


Figure 43 – Patrol guard with multiple behaviors in one state

However, this example unveils a problem : allowing multiple behaviors to be included in a single state, combined with the fact that behaviors may be finite (if their execution counter is not equal to *∞), may lead to certain behaviors finishing their execution earlier than others. Indeed, in *figure 43*, it is possible that either of the two behaviors in the *Friendly* state concludes before the other one.

Thus, a choice must be made : either allow behaviors to finish executing while the object remains within the state, or force a state transition when a behavior has terminated. The former leads to non-determinism : even when the current state of an object is known, it is impossible to determine which of the (finite) state behaviors are executing, which defeats the purpose of modeling behaviors inside states.

The latter approach implies adding a number of extra states, to make sure every behavior within a state is actually executing. It also means behaviors should be able to continue between states : for example, if the juggle behavior terminates before the patrol behavior, the object should transition into a new state. In this new state, the patrol behavior continues executing. This is what continued behaviors (section *4.8.2.2*) are used for.

Because the non-determinism of the former approach would in essence cause a breakdown of the entire state machine system – one would be designing states with unspecified contents – the latter one is adopted in VR-SDL. *Figure 44* shows the remodeled actor graph using this technique.

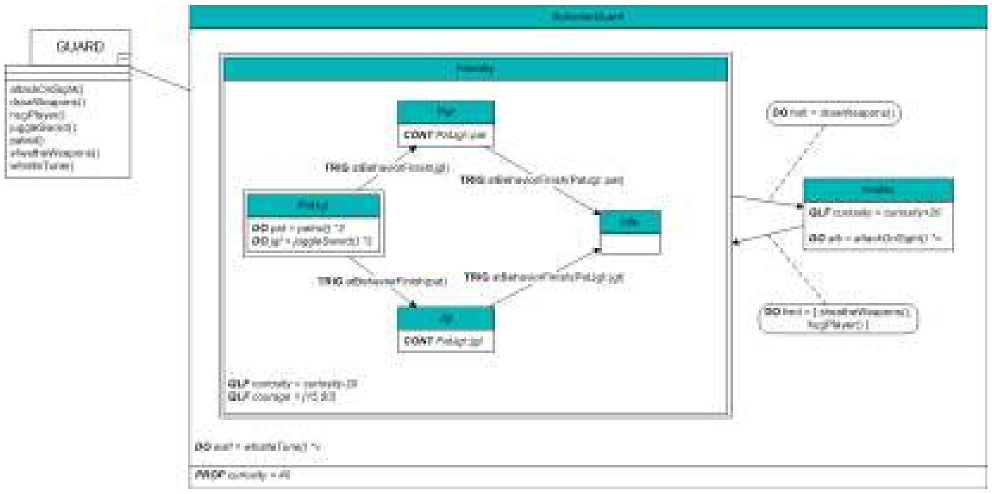


Figure 44 – Remodeled patrol guard with deterministic states

Note that the newly added states are irrelevant to the scenario : all that matters to the scenario is whether the guard is friendly or hostile. Therefore, it makes sense to model the new states as a subgraph of the original state, so the top-level states – and the transitions between them – remain unchanged. This also provides the possibility to leave qualifications or possible infinite behaviors in the encasing state.

Since behaviors might be modeled pseudo-randomly on the lower level (for example, the *juggleSword()* behavior could select a random juggle routine to perform from a number of predefined ones), it is impossible to know which of the state behaviors will finish first. Therefore, a state should be added for each combination of finished behaviors. One might expect this to lead to an uncontrollable explosion in the number of states, but this will never happen in practice : since all state behaviors in the same state should be simultaneously executable, the grand majority of the states will contain zero or one behaviors. And in the off chance that a state contains more behaviors, most of them will likely be small animations to increase realism (blinking eyes, twiddling fingers, scratching head, ...) which should be looped infinitely, and thus, do not cause the problem of early termination.

To conclude this section, consider the transitions in *figure 44*. They are labeled with trigger expressions (explained in section *4.10*), causing them to fire when certain behaviors finish. Within these triggers, the behaviors are referenced : notice that the behaviors in the leftmost triggers do not contain a scope delimiter, while the right ones do. This is an illustration of the scope of behaviors names : recall that the scope of a name is equal to the state it is defined in, plus that state's outgoing transitions. Hence, while the *pat* and *jgl* behaviors are defined within the scope of the transitions leaving the *PatJgl* state, they lie outside the scope of the *Pat* and *Jgl* states.

4.8.3.2. Modeling Transition Behaviors

Transition behaviors are quite similar to state behaviors, but without the main difficulties of the latter.

Unlike state behaviors, transition behaviors are executed only once : it is not possible to specify an execution counter (nor is it needed : 1 is the default amount of times a behavior is executed anyway). The state transition is only considered complete after all transition behaviors have been executed, until that point, the object is still in its previous state. Because of this, transition behaviors are supposed to be short, usually small animations or similar. Additionally, it makes no sense for a transition behavior to be defined using a `CONT` structure. After all, which behavior would be continued? Therefore, the only available notation for a transition behavior is the *`DO`* behavior.

Note that, like all `DO` behaviors, transition behaviors should be named. While they are never referenced inside an actor graph, this is needed from the viewpoint of the scenario graphs (as explained in section *5.7.2.1*).

4.8.4. Behavior Definition and Parameters

In the previous sections, an overview was provided of how to insert behaviors into an actor graph, and how they are executed. The actual definition of the behaviors was left largely unspecified, aside from mentioning that they should be modeled using lower level VR-WISE techniques (section *2.4*). In this section, the link to the lower level behavior modeling approach will be described in detail. First, single-actor behaviors will be described, consequently, more complicated behaviors with multiple actors are explained, and finally, some issues related to behaviors with reference actors are clarified.

4.8.4.1. Single Actor Behaviors

Recall from the figures shown earlier in this chapter that the available behaviors of the *GUARD* concept were shown in its drop-down box. The most important question to answer, is which behaviors are available to a certain concept.

To answer this, let us recapitulate the VR-WISE behavior modeling approach. It consists of two steps : defining the behavior (in terms of abstract behavior actors), and invocating the behavior. In this last step, the abstract actors are linked to either static concepts or instances thereof. In other words, once a concept is linked to a certain behavior through an invocation diagram, it is known that that concept can perform the behavior in question.

Considering that it is precisely one of these concepts which is linked to each actor graph, the answer to the question is straightforward : if a behavior modeling step is performed first, the behaviors that can be executed by a certain concept can be tracked by looking at the invocation diagrams containing the concept. In a sense, this provides an interface of all of the concept's behaviors. It is exactly these behaviors which are accessible for use in an actor graph (and consequently, which will be shown in the drop-down box). *Figures 45-47* illustrate this idea. Note that these are behavior diagrams, not actor graphs – although the notation used is very similar to that of actor graphs, its semantics are entirely different!

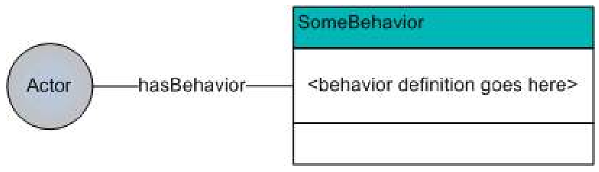


Figure 45 – Behavior Definition Diagram Schematic

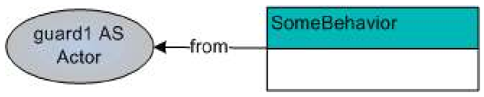


Figure 46 – Behavior invocation diagram schematic – instance

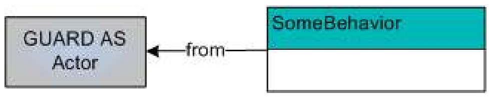


Figure 47 – Behavior invocation diagram schematic – concept

Figure 45 displays the definition of a behavior. At this stage, no behaviors are available to the *GUARD* concept. *Figure 46* invokes the behavior, by linking it to *guard1* (an instance of the *GUARD* concept). Still, the *GUARD* concept has no behaviors to choose from. Finally, in the invocation diagram in *figure 47*, the behavior's actor is specified to be of type *GUARD*. Now, *GUARD* does have 1 available behavior, namely *someBehavior()*. The semantics of this behavior are modeled by the diagram in *figure 45*, the actor is the guard object executing it.

Note the use of a Java/C++ method-like notation for the behaviors. This is done because behaviors are reminiscent of methods in an object oriented programming language. The notation is chosen to improve continuity with the property and state notations, which also used a Java-style syntax.

4.8.4.2. Multiple Actor Behaviors

Next, let us take a look at a more complicated behavior, with multiple actors, as shown in *figures 48* and *49*.

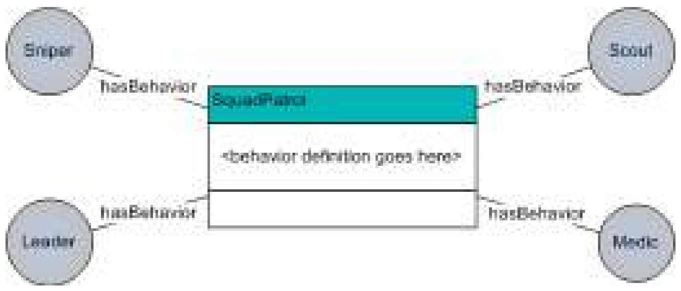


Figure 48 – Definition of a patrol behavior with 4 different actors

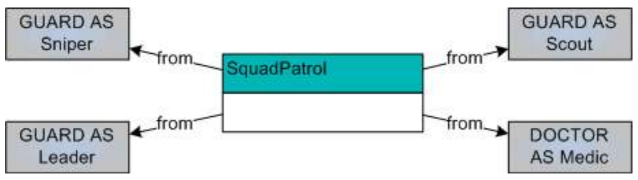


Figure 49 – Invocation of the behavior defined above

Since this behavior has multiple actors, things get a bit more complicated. If the behavior is rewritten in textual form, with the actors as parameters

squadPatrol(GUARD Leader, GUARD Sniper, GUARD Scout, *DOCTOR Medic)*

It becomes apparent that a single guard can play three different roles in this behavior (leader, sniper, or scout of the patrol). The fourth role, the medic, can not be played by a *GUARD*, but only by an instance of the *DOCTOR* concept.

Thus, an initial hunch could be to view an invocation diagram like the one in *figure 49* as the definition of three new behaviors for the *GUARD* concept : one for each of the roles played by the guard. Using this approach, each of these behaviors would be used in a separate actor graph, and thus, four actor graphs would be constructed : one for a *LeaderGuard*, one for a *SniperGuard*, one for a *ScoutGuard*, and one for a *MedicDoctor*. This is illustrated in *figure 50*. Note that for demonstration purposes, the name of the behavior was adjusted to reflect the role played by the object.

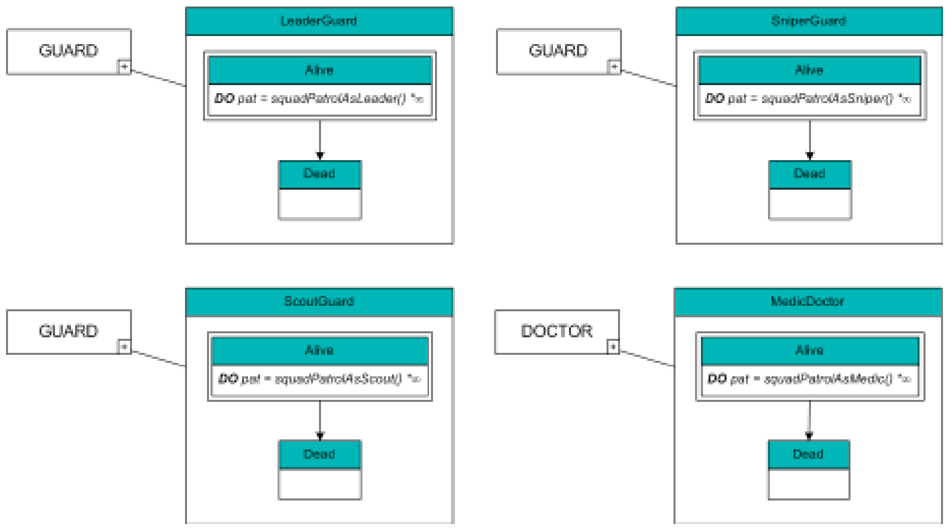

GUARD
LeaderGuard
Alive
DO pat = squadPatrolAsLeader() *∞
Dead
GUARD
SniperGuard
Alive
DO pat = squadPatrolAsSniper() *∞
Dead
GUARD
ScoutGuard
Alive
DO pat = squadPatrolAsScout() *∞
Dead
DOCTOR
MedicDoctor
Alive
DO pat = squadPatrolAsMedic() *∞
Dead

Figure 50 – Example actor graphs including the squadPatrol behavior

However, this approach has a severe problem : when each of these graphs is initialized as a scenario actor in a scenario graph, the *pat* behavior in each of the actor graphs would represent the same instance of the behavior, but with a different role being played by the current object.

Sadly, there is no way to know that the behavior is the same in each of the actor graphs, and thus, that it requires coordinated execution. This can lead to problems such as the *LeaderGuard* object being killed, prompting the behavior to break down : with the leader gone, the behavior can not be executed further. Events like these should be factored into the actor graphs (for example, by defining a new *Regroup* state to transition to when the leader is dead), but this can not be achieved by using four separate actor graphs, as shown in *figure 50*.

Therefore, for behaviors in which multiple actors actively participate (such as the *squadPatrol()* behavior), the actors should be restructured into an unconnected complex object (section *2.3.2.3*). Using this technique, the *squadPatrol()* behavior can be defined as a behavior of the group of objects, and thus, its participants are automatically coordinated. *Figures 51* and *52* show example graphs for the complex concept squad and its subobjects. The *squadPatrol()* behavior is now correctly represented as a single behavior, in the *Patrolling* state.

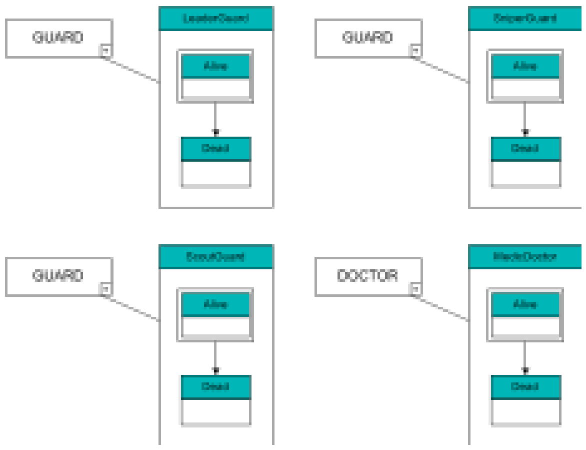


Figure 51 – Revamped actor graphs – the behaviors have all been moved to the Squad actor graph

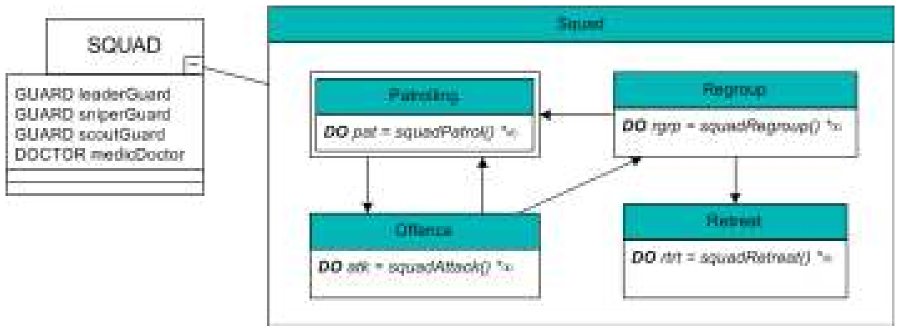


Figure 52 – Squad actor graph

4.8.4.3. Behaviors with Reference Actors

One final nuance remains : in the VR-WISE behavior modeling approach, it is possible to define an actor as enduring the behavior passively, rather than actively participating in it. These actors are called reference actors (section *2.4.1.2*). In the case where only one actor actively executes the behavior, while being dependant on multiple reference actors, the behavior is simply available to the concept of the active actor. The reference actors are added to the behavior as parameters, and the actual instances being used as reference actors in the execution of the behavior, are specified in the scenario graphs. This is explained in section *5.7.1.1*.

For example, suppose that a behavior would be defined where a guard points at some object. In this case, the object in question would be a reference actor in the behavior, and the behavior added to the actor graph would look something like *point(OBJECT target).* Of course, it is possible for multiple passive objects to be defined within a single behavior, in which case they are simply all added as parameters.

Take note that the only possible parameters in the behavior definition expression are actors. Other possible behavior-influencing variables, like speed or acceleration, are specified in the lower level behavior definition, and are hence un-adjustable within the actor graphs.

4.9. Visuals

4.9.1. General

It is possible that one would want to model an object which changes its appearance or sound between states. Consider a world with a castle, which can be occupied by an evil king or a good king. When the evil king has occupied the castle, its walls are black, its courtyards are neglected and in the background, an ominous violin tune sounds. When the good king takes over the castle, however, the walls swiftly turn white, flowers blossom in the courtyards, and the music changes to an upbeat lute melody. Suppose this is a game where smoothness of the gameplay is more important than realism, so the transformation between good and evil castle (and vice versa) should happen fluently, in a couple of seconds, giving the player a visual indication when a castle has just been conquered.

How would something like this be modeled? With the modeling techniques described so far, it is easy to see how the sound change would be designed : this is as simple as creating two states for the castle, *Good* and *Evil*, and add a separate (infinite) sound behavior in each state.

The graphical part however, is less obvious. There is currently no notation to specify this. Therefore, ***visuals*** are introduced : if a certain state is associated with a graphical representation of an object, then a visuals construct is added to the state. Such a construct consists of a keyword `VISUAL`, followed by the name of the graphics used. Per convention, graphic names are denoted like Java variables, to remain consistent with the rest of the notation. Hence, a visuals construct is denoted as :

```
<visual> ::= VISUAL <graphics-name>
```

States without a specified graphical representation use the concept's default one.

Figure 53 models the castle described above, using the visuals constructs. The transition behaviors morph between the views, and fade the music out. Should no animation have been specified, then the change in visual representation would have been abrupt and instantaneous.

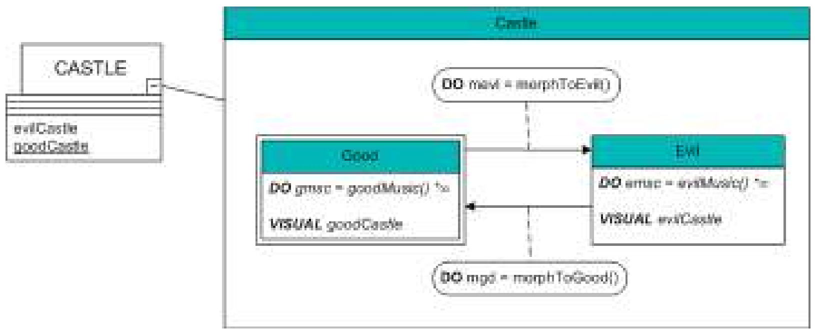


Figure 53 – Transforming castle, example of visuals

4.9.2. Views

An important question remains : visuals include the name of a graphical representation, but which graphics are available? Where are they modeled?

The most logical answer to this is, of course, at the lower level object modeling phase (section *2.3*). After all, this phase is about modeling the (graphical) properties of a certain concept. However, concepts contain far more information than just a graphical representation : they posses properties, and are linked to a number of behaviors, for example. Therefore, simply modeling a number of different concepts, with a slightly different look, and using these to represent the graphics, is unacceptable : the new concepts would have to be linked to the exact same behaviors and have the exact same properties as the original concept – in other words, they should be completely similar, except for the graphics.

Thus, the most obvious solution is simply to introduce a new construct in the object modeling phase, namely ***views***. A view of a concept is nothing more than a graphical representation for it, and a single concept can have any number of views assigned. Consequently, properties would still be defined once, for the concept, and behaviors should still be linked exclusively to the concept, while the concept itself could have several graphical variations. A concept always has a default view, representing its standard looks. Unsurprisingly, it is this view which is used as a graphical representation for state without specified visuals.

While views are not explicitly present in the VR-WISE object modeling phase, all the necessary constructs are available. Recall from section *2.2* that domain and world mapping ontologies are used to link concepts from the VE's domain to graphical representations. Views would simply be defined as mappings in either one of these ontologies : the default view is a mapping in the domain mapping ontology, while additional views may be defined as mappings in the world mapping ontology.

It is not hard to come up with examples demonstrating the use of views : modeling different types of chairs, for example, which have the same basic buildup (four legs, a sitting board and a back support board). Or the *GUARD* concept, mentioned numerous times throughout this chapter, could be given views for representing several types of wounded guards (missing a limb, sporting blood stains in various places, ...).

Like any relevant element modeled on a lower level than scenario modeling, views can be displayed in the concept's drop-down box (as shown in *figure 53*). Here, the default view is underlined[12].

4.10. Transitions

Up until now, all examples in this chapter used only one-way transitions. In the current section, different types of transitions will be introduced to make the designer's life easier, and the diagrams less cluttered. Additionally, the concept of triggers is explained.

Transitions are used to model the connection between states : if a transition from state A to state B exists, this means the object can switch to state B from state A. There are several types of transitions :

- ***One-way transitions*** indicate a transition which is possible from one state to another, but not the other way around. They are denoted as a single arrow. This is by far the most common type of transition.

[12] But remember that no formal specification exists for the notation of the contents of the drop-down box, since they are not modeled here, yet merely displayed.

- ***Two-way transitions*** (drawn as a line with arrowheads at each end) indicate a transition which is possible in both directions between two states. They are a shorthand notation for two one-way transitions between the same two states (one in each direction).
- ***Branching transitions*** are used when many states are interconnected (each state should be reachable from every other state). Instead of using a number of two-way transitions, which would quickly lead to cluttering of the graph, a dot is used, connected to each state through a single arrow. The name ‘branching transition’ is chosen because the dot essentially functions as a branching point : at this point, any of the lines can be followed to reach the next state. Note that this does not mean that the graph transitions into multiple states at once (this would be nonsensical), but rather, that the next state can be chosen from a number of options.

Figure 54 provides an example of the notation for each transition. Pictures a and b have identical semantics, picture c shows a partial graph for a guard with 5 moods, which can randomly be changed between.

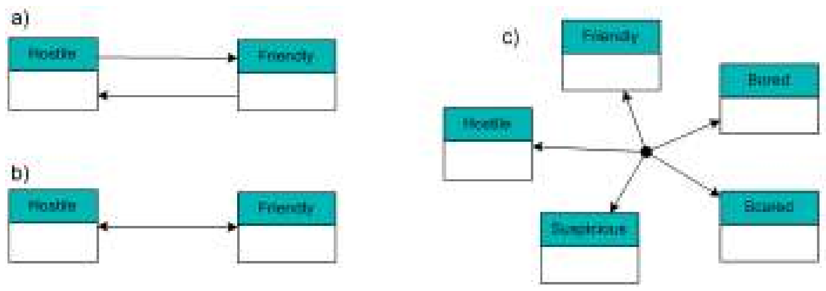


Figure 54 – Transition notations: (a) one-way transition; (b) two-way transition; (c) branching transition

In general, transitions only specify the *possibility* of a state change. The event triggering the state change is usually modeled within the scenario graphs, particularly for top-level transitions (transitions connected to at least one non-nested state). However, for one-way transitions, it is possible that a trigger is specified on the transition itself, enabling the designer to create more detailed objects, which change state automatically, based on some condition. These triggers are marked with a `TRIG` keyword. There are four different kinds :

- ***Behavior finish trigger*** **:** trigger fires when a certain behavior within the start state finishes execution. Notation :

```
<behavior-finish-trigger>  ::=  TRIG  atBehaviorFinish(  <behav-
ref> )
```

- ***Property qualification trigger*** **:** trigger fires when a certain property qualification becomes true. Unlike in section *4.7.1*, the property qualifications used here are not assignments, but rather comparisons of the property value. Properties could for example be manipulated by behaviors within the states, or by the scenario graph, causing the trigger to be activated. Notation :

  ```
  <prop-qlf-trigger> ::= TRIG <prop-ref> <comparator> <num-value>
  ```

- ***State qualification trigger*** **:** for complex objects, a change in state of one of the subobjects might prompt a state change of the entire object as well. Notation :

  ```
  <state-qlf-trigger> ::= TRIG <subobject-ref> : <state-ref>
  ```

- ***Elapsed time trigger :*** trigger fires after a set amount of milliseconds. Notation :

  ```
  <elapsed-time-trigger> ::= TRIG onElapsedTime( <num-value> )
  ```

On a final note, it is possible (though extremely rare) that a transition could be triggered in several ways, and multiple triggers should be specified. This is achieved by separating the triggers using an `OR` keyword.

4.11. Inheritance

Consider two different types of guard : one is the regular patrol guard, mentioned numerous times in this chapter, who is either friendly or hostile towards the player. The other is a cowardly guard, who patrols along his route, like a regular patrol guard, but instead of turning hostile and attacking the player, he gets scared and runs away instead. Both guards have a *Friendly* state which is entirely similar, and identical transitions to a second state. The only difference is this second state itself.

To the reader familiar with object oriented programming languages, this will immediately sound familiar : indeed, it sounds remarkably similar to inheritance in OO programming. The concept of defining a new graph as a slightly altered version of an existing graph is one which sounds attractive for scenario modeling : while it requires little extra work to model the example above as two separate actor graphs, it would be considerably more laborious to model if the guards had thirty matching states, and one divergent one. Therefore, inheritance is included in the VR-SDL object modeling approach.

4.11.1. General Approach

To define an actor graph as a child of another actor graph, a UML-like notation is used (a line with a triangular white arrowhead, going from the child graph to its parent). The child graph inherits everything from the parent graph, including states, their contents, and transitions – except when specified otherwise. Therefore, everything bar that which changes can be omitted from the child graph.

For example, suppose that a child graph is defined, identical to its parent graph, with the exception of one altered state. Let us call the original state (in the parent graph) *S*, and the new state (in the child graph) *S'*. Then *S'*, called an ***overwriting state***, should have the form *newName:S*. Qualifications, behaviors and visuals from S do not propagate over to *S'*, and thus, *S'* effectively starts as a blank state : if it should contain properties, behaviors or visuals that were also present in *S*, they need to be written down explicitly in *S'* as well.

If *S'* is the only thing that changes compared to the parent graph, while all other states and transitions remain identical, all that needs to be included in the child graph is the changed state. *Figure 55* shows an example actor graph for the guards described above.

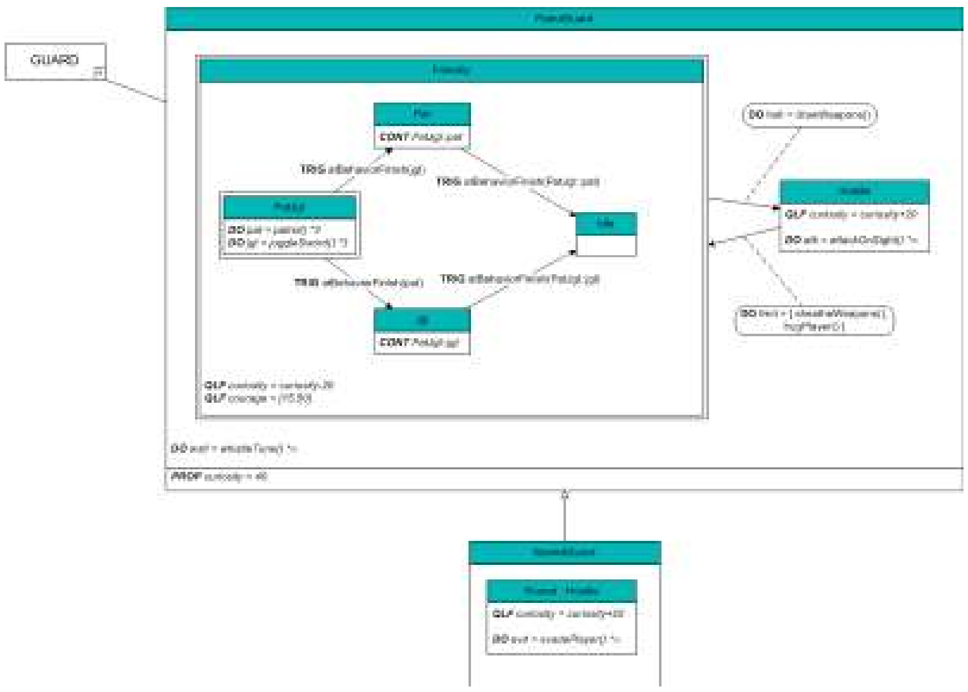


Figure 55 – Inheritance example : cowardly guard

If the full *CowardGuard* graph were to be drawn out, it would look exactly the same as the *PatrolGuard* graph, with the exception of the *Hostile* state being replaced by the *Scared* state. Note that, although the scared state contains the exact same qualification as the *Hostile* state, it should still be added explicitly.

Note that the child graph needs not be linked to a concept, since it is obligatory for it to be linked to the same concept as its parent graph. This is because the parent graph may utilize numerous characteristics specific to its linked concept (properties, behaviors, ...), which means the child graph should know these as well. This can only be acquired by linking it to the same concept as its parent graph.

When modeling inheritance, care should be taken that an overwriting state does not delete items which are referenced somewhere else in the graph (such as a behavior which is continued in another state). However, this is the responsibility of the designer.

4.11.2. Dummy States

The approach in the previous section works fine if the only changes take place inside the states. In other words, if the triggers on the transitions are the same as in the parent graph, if there are no extra or fewer transitions, if the transition behaviors are identical, and so on. Naturally, there is no guarantee that this will always be the case. Therefore, it is allowed to add ***dummy states***, and specify the transition between the dummy state(s) and the overwriting state(s).

A dummy state is simply an empty state (inside the diagram), all contents of the state (behaviors, properties, etc...) are identical to the parent graph. There is no risk confusing a dummy state and an overwritten state, since dummy states always have the same name as a state from the parent graph, while new states follow the naming convention described earlier.

If a dummy state is added to the child graph, all transitions between it and other dummy states, or states which are not mentioned in the child graph, are identical to the parent graph. Hence, there is no need to draw them. On the other hand, any transitions between dummy states and overwriting states become void. If a transition between a dummy state and an overwriting state was already present in the parent graph (in the state that was overwritten), then it still needs to be redrawn in the child graph, along with any new or changed transitions.

The reason for this lies in the removal of transitions. The alternative to the former approach is to provide a notation for the removal of a transition, and only include adjusted transitions in the child graph (in this context, ‘adjusted’ means added, deleted or changed). However, this would lead to a child graph in which certain visible transitions would only be drawn to signify that they are not really there, while other transitions would be drawn because they are there now, but were not before. Even with a clear notation, this would be highly unintuitive, and thus, the former approach is used in VR-SDL.

Figure 56 provides an example of dummy states. It shows the scared guard from before, but now, the guard can no longer revert to the *Friendly* state after getting scared. This is achieved by removing the transition from *Scared* to *Friendly*, using a dummy *Friendly* state.

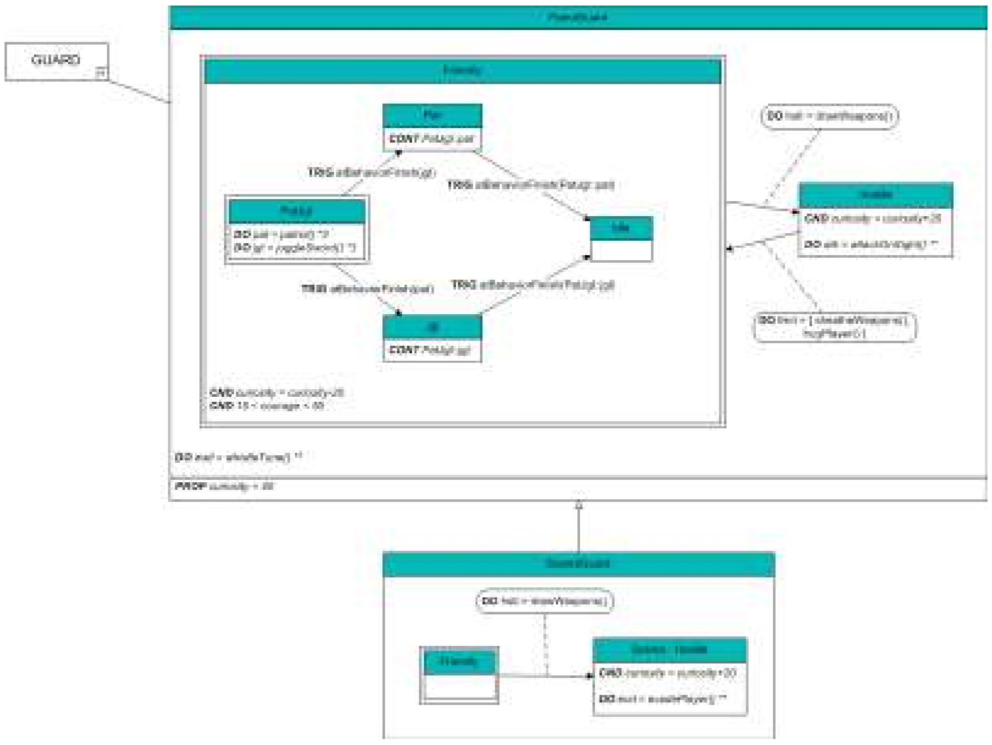


Figure 56 – Remodeled cowardly guard, using a dummy Friendly state to model an omitted transition

The only thing which might be difficult to express with the current notation, is the removal or modification of a transition between dummy states. As it stands, a changed transition is easy enough: the transition is simply added, with the appropriate alterations (new behavior, new trigger, different type of transitions, ...) A removed transition can also be modeled, but it requires a bit of a detour. One of the dummy states connected by the transition should be changed to an overwriting state (and consequently, all properties, behaviors and such should be re-entered), and all transitions between the two states, excluding the one which should be deleted, need to be redrawn.

This is a smaller problem then it may at first glance appear to be, since it is unlikely to happen that a transition between two unchanged states should be changed or deleted. Most likely, if a transition changes, one of the states it connects to will change as well.

4.12. Conclusion

In this chapter, a concise overview of the modeling constructs for actor graphs in VR-SDL was given. Actor graphs are used to model the relevant states for the types of actors participating in a scenario. Their main components are :

- **Linked concept :** The lower level concept linked to the actor graph. This provides a link to the object modeling phase in VR-WISE.
- **States :** The main building blocks of an actor graph. States are used to model the changes that might occur in the actors during the course of a scenario. They can contain qualifications, behaviors and visuals. An actor graph has one encasing state, called its frame.
- **Properties :** Mutable assets of an actor graph.
- **Qualifications :** Specify a value for a property or state for a subobject, while the object is in a certain state.
- **Behaviors :** Signify that the object is performing some behavior(s) while in a certain state, or when switching from one state to another. The behaviors are defined on the lower level, using the VR-WISE behavior modeling technique.
- **Visuals :** Declare a look for the object while it is in a certain state.
- **Transitions :** Forms the connection between the different states, detailing the possible state changes. They may or may not be labeled with a trigger : in the former case, the actor graph is in control of when the transition fires, while in the latter case, this is the responsibility of the scenario graph.

Additionally, an inheritance notation is included for modeling similar types of actors.

Once the necessary actor graphs for a scenario have been designed, they can be used to define the scenario's actions. In the next stage of the scenario modeling, a scenario graph will be constructed, specifying the possible courses of the scenario, and the influence they will have on the actor graphs of its actors.

Chapter 5

Scenario Graphs

5.1. Introduction

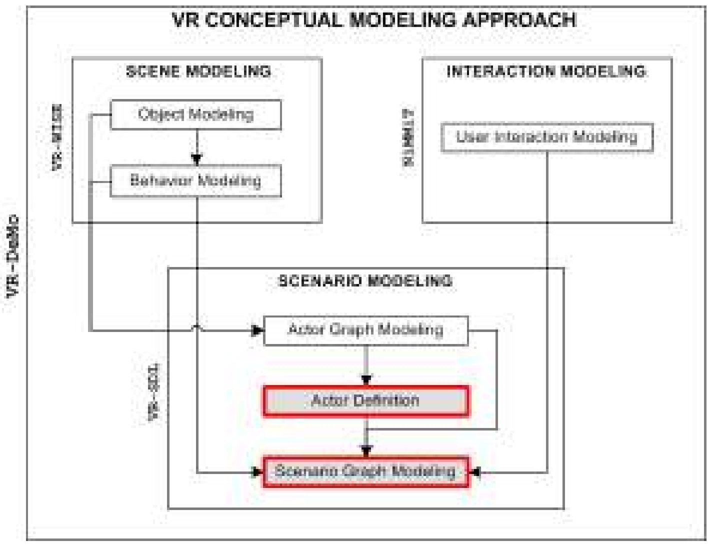


Figure 57 – Scope of this chapter : *actor definition* and *scenario graph modeling*

In the previous chapter, *actor graphs* were described as a way to model the actors which can be manipulated during the scenario. However, the scenario itself is yet to be modeled. In VR-SDL, this is done by using *scenario graphs* (*figure 57*). A detailed overview of their components and notation is provided in this chapter.

Scenario graphs are a kind of flowcharts, consisting of a web of linked events – actions performed by the user or an actor – defining the possible flows of the scenario. They provide facilities to formally define the amount and type of actors in the scenario (*figure 57*), and to manipulate these actors during the course of the scenario (through pre- and post-conditions of the events).

This chapter opens with an example scenario, which is used as a recurring theme throughout the explanation of the different concepts involved in modeling the scenario graph. Consequently, a general overview of the approach and different constructs for modeling a scenario is given. After that, a closer look is taken at the different parts of a scenario graph.

Like the chapter on actor graphs, the bulk of this chapter consists of a very in-depth description of the scenario graph notation, issues with scenario modeling, reasons behind the introduction of several components, and so on. Therefore, the reader who is not concerned with these details, and is mostly interested in getting a general overview of the approach, may skip most of the technicalities and limit himself to sections *5.2, 5.3,* and *5.10*.

5.2. An Example : Fire Alert

Before turning to the explanation of the scenario graph notation, let us consider an example scenario, describing a fire alert in a building. This example will be used throughout the chapter, to demonstrate the different constructs involved in the modeling of scenarios. A full VR-SDL specification of the scenario (including actor graphs for the actors) is presented in appendix *C*. Chapter *6* returns to this example, reiterating the entire VR-SDL approach.

The *fire alert* case study is a small game, where the user plays the role of a fireman tasked with rescuing as many workers as possible from a burning office building. The building has two floors, each containing a hallway and five offices : the first floor is accessible through the building's front door, while the second floor can be reached by climbing a fire escape at the side of the building.

The user takes control of a 3D fireman avatar, carrying a portable fire extinguisher with an unlimited supply of fire repressing foam. Before commencing play, the user may alter his fireman's appearance by selecting a face type and skin color from a menu. Once this has been done, he is given control of the fireman and must enter the burning building to get the workers out.

The player can choose whether he enters the building via the front door, or via the fire escape. In the former case, he needs to extinguish the entrance first, which may cause the corroded door to collapse. If this happens, the rubble needs to be doused and cleared before the front entrance is available again. In case the fire escape is selected as the means of entry, there is a chance it may collapse under the fireman's weight. Since the player does not have the necessary tools at his disposal to repair a broken fire escape, an occurrence of this event would leave only the front door available for entering the building.

Once inside, the player can look for trapped workers, either in the hallways or in the offices. Since all the office doors are burning, they need to be extinguished in order to grant access, which may once again lead to a cave-in, similar to that of the front door. However, inside the building, there is not enough room for clearing a collapsed door. Therefore, these rooms become permanently unavailable.

When a worker is encountered, either in a hallway or in an office, the player must talk to it and agree to escort it out of the building. There are two options for achieving this : either the worker may be lead out of the front door, or the fireman may extinguish and break an office window. Then, he can attach a rope to it so the worker can climb down.

The fireman has a final ace up his sleeve : he has one experimental foam bomb in his backpack, which may be used inside an office, to extinguish it entirely. However, due to his unfamiliarity with the device, he must first build up his confidence level by rescuing enough people, before being able to deploy it.

5.3. Approach

Recall from the VR-SDL methodology explanation in section *3.4* that modeling a scenario consists of four steps (remember that step 1 is informal, and thus does not show up on the diagram in *figure 57*) :

1. **Description step :** Informally describing the scenario and the different kinds of actors participating in it.
2. **Actor graph modeling step :** Modeling actor graphs for the different types of actors.
3. **Actor definition step :** Formally defining the actors.
4. **Scenario graph modeling step :** Modeling the scenario graph.

Step three and four constitute the modeling of the actual scenario, as described in this chapter.

Although the actor definitions (section *5.4*) are not physically a part of the scenario graph (they are merely written down in a textual form), they are very closely tied to the scenario graph modeling step. This is because the actor definitions specify the name by which the actors will be referenced in the scenario graph, as well as their role (an actor graph), which determines their possible evolution during execution of the scenario. Hence, the actor definitions contain plenty of information which is used in – and thus essential to – the scenario graph.

Scenario graphs themselves are a special kind of flowchart. A scenario is described as a web of ***events*** (section *5.5*), drawn as rounded rectangles, which are connected to their possible follow-up events through ***transitions*** (section *5.8*). There are two types of events : events describing an action taken by the user or one of the actors in the scenario (section *5.6*), and events used for enhancing the graph structure and expressing additional information, unrelated to one particular action.

Both kinds of events may contain ***conditions*** (section *5.7*), which are used for two purposes : pre-conditions specify requirements that must be fulfilled for the event to be able to occur, while post-conditions describe side-effects of an event. The latter can be used for triggering state changes in actors, altering the value of some actor property, and more. Additionally, ***terminals*** are introduced to denote points of no return in a scenario (section *5.9*).

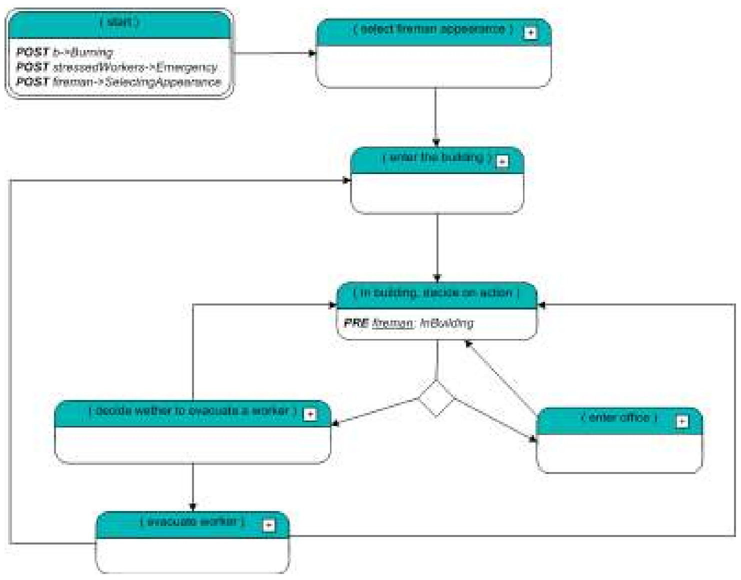

(start)
POST b->Burning
POST stressedWorkers->Emergency
POST fireman->SelectingAppearance
(select fireman appearance)
(enter the building)
(in building, decide on action)
PRE fireman: InBuilding
(decide wether to evacuate a worker)
(enter office)
(evacuate worker)

Figure 58 – Top-level scenario graph for the fire alert example

In *figure 58*, the scenario graph notation is illustrated by means of the top-level scenario graph for the *fire alert* example. The events containing a plus-sign in their upper right corner can be expanded to view the more detailed models, but the general flow of the scenario is clearly visible : at the start, the building (called *b*) is burning , the workers are in a state of emergency and the fireman's appearance is being selected. After finishing this, the player enters the building, and decides on the action to take : either enter an office, or evacuate a worker. After completing the former tasks, he needs to select an action again, while completing the latter may also leave him outside the building, prompting him to re-enter it.

Figure 59 shows a graphical overview of the different components described in this chapter, labeled with the corresponding section they are described in. Note that these could be substituted for the grey boxes in *figure 57*.

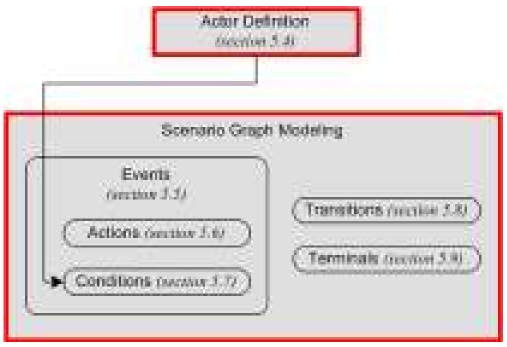


Figure 59 – Actor definition and components of a scenario graph

5.4. Actors

As explained in chapter *3*, and repeated in section *5.3*, the actors participating in a scenario need to be formally defined before the actual scenario graph is modeled. This is logical, of course : not defining the actors before the scenario graph is created would be like filming a movie before the roles have been cast.

This section provides a detailed overview of a number of issues related to actors in a scenario graph. Section *5.4.1* describes how actors are defined, while section *5.4.2* describes how to reference them. Consequently, two additional modeling issues concerning actors are explained : actor subgroups (section *5.4.3*) and user-controlled actors (*5.4.4*).

5.4.1. Defining Actors

While the actor definitions are strongly coherent with the scenario graph, they are not a physical part of it. Instead of using some graphical notation, linked to the actual graph, the actor definitions are written down in a text form. There are multiple reasons for this decision :

- While the actors participate in the scenario, their definition is not a part of it. Therefore, it makes little sense to include it in the actual graph.

- Because actor definitions are very short and simple, the benefits of a graphical notation over a textual one are negligible. In fact, the extra space taken up by a graphical notation would vastly outweigh the increase in readability (if any).

There are two types of actor definitions : ***single actor definitions***, and ***actor set definitions***. Both types contain two mandatory segments : the role played by the actor, and its name. Additionally, actors may be instantiated by linking them to physical objects from the VE.

5.4.1.1. Single Actor Definition

A single actor definition has the following form :

```
<single-actor-definition> ::= ACTOR <actor-name> role <actor-role>
                                        playedBy <instance-ref>
                            | ACTOR <actor-name> role <actor-role>
```

Note that actor definitions always contain a *name* and a *role*, and possibly an *initialization*. Two example actor definitions (one initialized, and one non-initialized) are given below :

```
1. ACTOR fireman role Fireman playedBy fireman1
2. ACTOR bomb role FoamBomb
```

Each actor has a ***name***, which is used to reference it in the scenario graph. Actors also have a ***role***[13] : this is defined by an actor graph, which models its behavior during the course of the scenario. The available role names are the names of all the actor graphs defined in the *actor graph modeling step*. Hence, for the examples shown above to be valid, the actor graphs in *figure 60*[14] need to exist. Note that the role in the first example, *Fireman*, is the name of the actor graph in *60a*, while the role in the second example, *FoamBomb*, is the name of the actor graph in *60b*.

Observe that by linking an actor to an actor graph (through its role), it is indirectly linked to a concept as well (the actor graph's *linked concept*). In the examples above, the concept of the *fireman* actor is *FIREMAN*, and the concept of the *bomb* actor is *BOMB*.

[13] Not to be confused with role names in complex concepts (section *2.3.2.2*).

[14] The full definition of these actor graphs is presented in appendix *C*.

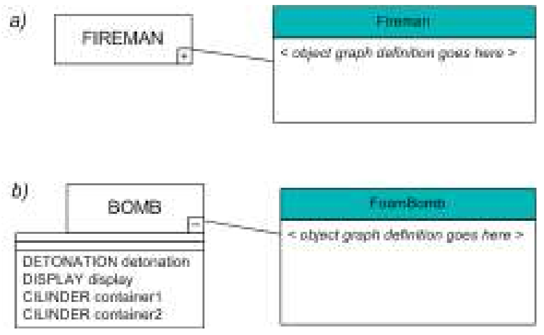


Figure 60 – Example actor graph frames for the fire scenario

So far, actors have been described in a rather abstract way : they have a *name*, a *role* and (through the role) a *concept*. Essentially, this provides enough information to model the entire scenario graph : it is known what actors are called, how they will behave, and how they are constructed. However, one final part is still missing from the actor definitions : the abstract actors should be made concrete by linking them to concrete objects (or *instances*, as described in section *2.3*) from the VE. This process is called ***initializing*** (or *instantiating*) the actor. Obviously, actors can only be linked to instances of the correct concept.

To understand what it means to initialize an actor, consider the *fireman* actor. It is possible that the VE in which the fire alert scenario will take place, will have multiple *FIREMAN* objects. Each of these objects have an id to reference them in the world, such as *fireman1*, *fireman2*, and so on. By initializing the *fireman* actor, one of these FIREMAN instances is now selected to play the role of the fireman in the scenario – in this case, *fireman1*. The rest may be used as actors in different scenarios, or not at all.

Note that actor initialization is not mandatory. Initialized and non-initialized actors are interpreted differently in the scenario : when an actor is initialized, it means that it is available from the start of the scenario. Non-initialized actors are actors that do not yet exist when the scenario commences, but could be added at run-time. Note that this implies that every actor which might at some point take part in the scenario, is defined at the start. In the *fire alert* example, the foam bomb is an example of an actor that could be defined as non-initialized : the foam bomb is only created in the VE, when the player decides to place it in some room. The fact that the player has the *possibility* to place a foam bomb, however, means that it should be added to the list of actors from the start – albeit not initialized.

Now that the components of an actor definition have been explained, let us return to the example definitions at the start of this section. Their interpretation is as follows :

1. Defines an actor called *fireman* of role *Fireman* (an actor graph), and initializes it by linking it to a *FIREMAN*[15] instance from the VE, namely *fireman1*.
2. Declares a non-initialized actor called *bomb*, of role *FoamBomb* (again, this is an actor graph). This means that at some point in the scenario, the bomb actor may (but does not have to!) be initialized (e.g. when the fireman places it).

5.4.1.2. Actor Set Definition

Actor sets are used to describe collections of indistinguishable actors within a scenario. This means that they all play the same role, and are in the same state at each point of the scenario (except when in a different *actor subgroup*, as explained in section *5.4.3*).

As shown below, an actor set definition is quite similar to a single actor definition.

```
<actor-set-definition> ::= ACTORS <set-name> role <actor-role>[Integer]
                                  playedBy [ <instance-list> ]
                         | ACTORS <set-name> role <actor-role>[]
```

Note the use of the keyword `ACTORS`, as opposed to `ACTOR` in a single actor definition.

The basic components of a single actor definition return in an actor set definition : actor sets have a *name*, a *role* (identical for all objects in the set), and are possible *initialized*. In addition, initialized actor sets also have a specified size – but non-initialized sets do not. Unlike with single actor definitions, non-initialized actor sets are not used to add newly created actors during the course of the scenario, but rather, to add already existing actors at some point. They are used for defining subsets with deviating properties from a basic actor set. As mentioned above, this concept is called an *actor subgroup*.

The initialization of actor sets is very similar to that of a single actor. The difference is that, instead of adding a single instance, a comma-separated list is used, delimited by square brackets.

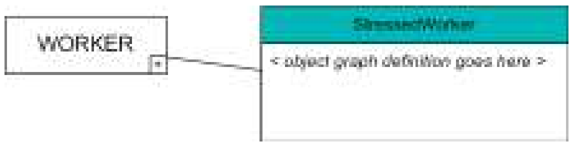


Figure 61 – Another example actor graph frame from the fire scenario

[15] Remember that this can be deduced by looking at the actor graph's linked concept.

With the actor graph from *figure 61* available, examples of both an initialized and a non-initialized actor definition are :

```
1. ACTORS stressedWorkers role StressedWorker[4] playedBy
          [w1, w2, w3, w4]
2. ACTORS rescuedWorkers role StressedWorker[]
```

These are interpreted as follows :

1. Defines a set of workers, called *stressedWorkers*. Each member of the set plays the role of *StressedWorker* (the actor graph shown above). The set has four members, and contains the objects *w1*, *w2*, *w3*, and *w4* from the VE. Each of these is an instance of the *WORKER* concept.
2. Defines an empty set of workers, called *rescuedWorkers*. During the course of the scenario, actors of type *StressedWorker* may be added to this set. Note that actors can be in multiple sets at once, so members of *stressedWorkers* may (and will) be added to *rescuedWorkers* at some point. Also observe that no size is specified here, this is not allowed, as the set must remain dynamic – it is not known in advance how many actors will be added to the set.

5.4.2. Referencing Actors in Scenario Graphs

Once the actors participating in a scenario have been defined, they are available for use in the actual scenario graph. To indicate that an actor changes under the influence of a scenario, one must of course be able to reference it therein.

As mentioned before, this is what actor names are used for. In the simplest case, for single actors, an actor can be referenced by simply using its name. The first example actor from section *5.4.1.1* can be referenced as *fireman*.

In addition, if an actor's concept contains subobjects, these can be referenced in the same way as in the actor graph specification : by using a dot-notation. In other words, if a foam bomb is modeled as shown in *figure 60*, its detonator could be referenced by using *bomb.detonator*. Again, like in actor graphs, the concept names of the subobjects can be used to reference the set of all subobjects of a particular concept. The reference *bomb.CILINDER* would return a set with the *container1* and *container2* subobjects of the *bomb* actor.

Referencing an entire set of actors (for example, to denote a state change for all of the actors in the set) is entirely similar to referencing a single actor : the set name is used to reference it. Obviously, since sets may not contain subobjects, it is impossible to reference subobjects from a set.

Referencing a single actor from a set is a bit more involved. Recall that, from the viewpoint of a scenario, actors in the same set are undistinguishable from one another. Therefore, it does not make sense to use an index to access one actor from a set (e.g., *stressedWorkers[2]*) – if this were required, the actors would have been defined separately. The specifics of referencing an actor from a set are explained in section *5.5.2.2*.

5.4.3. Actor Subgroups

As stated in section *5.4.1*, it is possible that one would want to reference a subset of the actors from the same set, instead of the whole set. For example, there are a number of stressed workers participating in the fire scenario, but the ones among them that have been rescued by the player will obviously behave differently from their counterparts which are still trapped in the building. This implies that some of the stressed workers should be in a different state than others.

To tackle this issue, a new concept is introduced: ***actor subgroups***. An actor subgroup is a group of actors with the same role. The *sub-* prefix follows from the fact that all actors with the same role will initially be defined as one set (such as the stressedWorkers set from section *5.4.1.2*), so an actor subgroup is essentially a subset of this initial set.

Subgroups are entirely dynamic : during the course of the scenario, actors could be added to or deleted from the subgroup, and it is unknown how large it will be at any point in the scenario. The addition and deletion of actors from the subgroup is discussed in section *5.7.2.4*. Here, the focus lies on how to define actor subgroups.

The possible actor subgroups should be known before executing the scenario – thus, while modeling its graph. In a sense, these subgroups define the different conditions an actor might endure in a scenario. Obviously, these should be known from the start, otherwise they could not be modeled. For instance, returning to the *fire alert* example, two sets would be defined : *stressedWorkers* and *rescuedWorkers*. The first set represents the workers still trapped inside the building (and thus, it initially contains all workers), while the second one – the subgroup – contains the workers that have been freed (and hence, it is empty at the start). As the scenario progresses, actors will be added to or deleted from the subgroup, based on the occurrence of some event.

The notation used for defining actor subgroups has already been declared in section *5.4.1.2* : a non-initialized actor set is used, specifying the role of the actors in the subgroup and the name used to reference it. Remember that a non-initialized actor set definition is not allowed to have a pre-specified size – this is because it is impossible to predict how the scenario will develop, and thus, how many actors will be added to the subgroup.

An interesting property of actor subgroups is that, although they are completely flexible in terms of membership (each actor of the correct type can be added to or deleted from the set at runtime), they can be referenced in the same way as regular actors can. In other words, one could say "all workers who have been rescued are cheering" by declaring that all actors in the set must be in the *Rescued* state, wherein a cheering behavior is recorded. Consequently, when a player rescues a worker, he would be added to the *rescuedWorkers* subgroup.

5.4.4. User-Controlled Actors

While scenarios can be defined without any form of user interaction, this would not prove to be very interesting : In such a case, the scenario would consist of a predetermined series of events, with the only possible variation being based on pure randomness – by haphazardly selecting a next event from a number of possibilities. Clearly, user-interaction is required to provide for more appealing scenarios.

In general, the user can interact with a scenario in two ways, both of which have some impact on the definition of actors. Primarily, there is ***avatar user interaction***, where the user manipulates the VE through some actor (his *avatar* – hence the name). Secondly, there is ***interface user interaction***, where the user represents the actual person sitting behind the computer[16], manipulating the VE through menus and widgets which are not physically part of it.

Note that it is possible for multiple users to interact with the application at the same time. This case is not considered here, and is left for future work. In the remainder of this chapter, user interaction always assumes a single user.

[16] Or in more advanced VR environments, the person wearing the visor and sensor gloves.

5.4.4.1 Avatar User Interaction

Avatar user interaction represents all interaction the user has with the actual virtual environment, through the use of his avatar – in the *fire alert* example, this is the fireman. For example, the user can not extinguish a fire himself, but the fireman actor can. An actor functioning as the user's avatar is modeled and defined in the exact same way as a regular actor, with one minor difference : when defining or referencing the user's avatar, its name is underlined. The slightly altered fireman actor definition now looks as shown below :

```
ACTOR fireman role Fireman playedBy fireman1
```

At any point during the execution of a scenario, the user has at most one avatar. This makes sense, as the user can only control a single object at a time. However, the avatar is not limited to a single actor, it can be an actor set as well. The latter is possible because all the objects in the set will perform the exact same actions. Hence, the user is really just entering a single stream of actions, which happen to be performed simultaneously by a number of different actors. An example of this are real-time strategy games like the recently announced *Starcraft 2* by *Blizzard Entertainment*[17], where the player can select a random number of units of a certain type, and give them instructions like moving to a certain spot, or killing a group of enemies.

Additionally, it is possible for the user to get a different avatar during the course of the scenario. Consider a squad-based shooting game[18], for example, where the user commands a group of four soldiers, each with their own specific skills. At any point during gameplay, the player has direct control of exactly one soldier, while the other three are being steered by AI. The player can freely switch between the four soldiers, to maximally utilize their individual capacities.

Due to this possible change in avatar during the execution of a scenario, it is imperative that the actor which currently serves as the avatar is clear at any point in the scenario. Therefore, at definition, and whenever the avatar is referenced, its name is underlined.

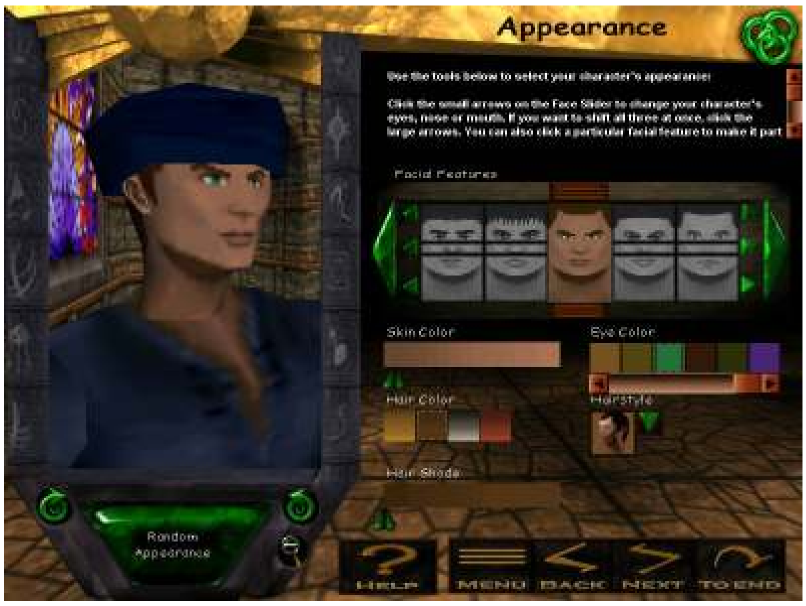
5.4.4.2. Interface User Interaction

In the *fire alert* scenario, an example of interface user interaction would be the start, where the user selects a face style and skin color for his fireman. In the actual application, this could be done through a pop-up menu which is brought up by

[17] *http://www.starcraft2.com/* (access date 19/05/2007)

[18] Such as the popular *Conflict* games : *http://www.conflict.com/* (access date : 30/5/2007)

right-clicking the fireman, by using a menu which automatically appears when the scenario commences, and so on... *Figure 62* illustrates such a character customization interface.


Use the tools below to select your character's appearance!

Click the small arrows on the Face Slider to change your character's eyes, nose or mouth. If you want to shift all three at once, click the large arrows. You can also click a particular facial feature to make it part

Figure 62 – Character customization menu from the game Asheron's Call[19]

In this case, the user is clearly not interacting with the VE through an avatar, but rather, he is interacting with a menu that is not actually part of the virtual environment, but provides information about it.

In scenario graphs, the actor name *user* is reserved for modeling interface user interaction, and is thus unavailable for naming regular actors. No definition is specified for the *user* actor, since it has no physical representation in the VE, and thus, it needs not be linked to an actor graph either. Its sole purpose is to describe active events (section *5.5.2*) taking place on the interface level, as human-computer interaction. Like the avatar of the user, any time the *user* actor is mentioned, its name is underlined.

5.5. Events

While actors are essential to the interpretation of a scenario graph, their definitions are not actually a part of it. As described in section *5.3*, a scenario graph is

[19] Picture from *http://acvault.ign.com/* (access date 19/05/2007)

basically a sort of flowchart, connecting events in the world to possible follow-up events, thus creating a web of actions that may influence and lead to one another.

This section describes these events. In section *5.5.1*, a general overview of the event notation is provided. Afterwards, two different types of events are described : *active events* in section *5.5.2*, and *passive events* in section *5.5.3*.

5.5.1. Notation and Contents

5.5.1.1. Event Notation

Events display many similarities to actor graph states :

- The notation for events is similar to that of states in actor graphs, namely a double rectangle, with the event name in the top box and extra information in the bottom one. Equivalent to states, these boxes are called the event's ***name field*** and ***content field***, respectively.
- Like actor graph states, each (sub)graph for a scenario may contain a single start event, where the scenario execution starts. Start events are drawn with a double edge (again, in analogy with start states).

However, to avoid confusion between the concepts of state and event, events are denoted by rounded rectangles. *Figure 63* demonstrates the event notation.

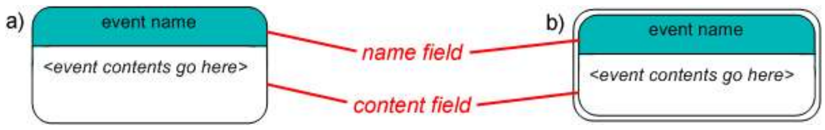


Figure 63 – (a) Event notation; (b) Start event notation

There are two main types of events : ***active events***, representing an action taken by some (possibly user-controlled) actor, and ***passive events***, used to clarify the graph structure and facilitate modeling. Both are described later, in sections *5.5.2* and *5.5.3*.

Notation-wise, the main difference between an active event and a passive event lies in the rules for writing the event name – in other words, in the *name field*. The *content field* displays both similarities and deviations between the two event types.

The *content field* of both active and passive events may contain conditions : primarily, there are ***pre-conditions*** – conditions which must be true for the event to be able to be executed – and secondly, there are ***post-conditions*** – results (or side-effects) of executing the event. The latter imply some change in the VE, such as an actor changing state. Note the subtle difference in interpretation of pre- and post-conditions : pre-conditions imply something *being* true before entering the event, while post-conditions imply something *becoming* true after exiting the event. A pre-condition can only hold if it is true from the start of the scenario, or if some event has occurred which has it as a post-condition. There are numerous different types of both pre- and post-conditions, all of which are described in detail later (section *5.7*).

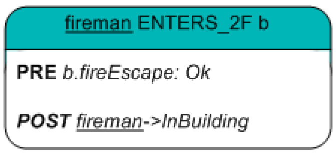


Figure 64 – An event containing one pre-condition and one post-condition

Figure 64 shows an event with both a pre- and a post-condition. Informally, this event models the fireman entering the building on the second floor, via the fire-escape. It states that, for this to be possible, the *fireEscape* actor (a subobject of the *Building* actor *b*, which was mentioned earlier in the description of *58*) should be in the *Ok* state – in other words, the fireman can not enter the building using a collapsed fire escape, which makes sense. Additionally, after the event is executed, the *fireman* actor will be in the *InBuilding* state.

5.5.2 Active Events

This section goes into detail about active events. As mentioned previously, active events model some action being taken by an actor. Actions are written in the event's *name field*. The *content field* contains requirements and side-effects of the action, specified through *pre-* and *post-conditions*.

There are three different types of ***actions*** which can occur :

- An AI actor performs an action or interacts with another AI actor. For example, a worker in the *fire alert* scenario could spontaneously start talking to one of his coworkers.
- The user's avatar performs an action or interacts with an AI actor. This is *avatar user interaction*, as described in section *5.4.4.1*. In the *fire alert* scenario, every action taken by the *fireman* actor is an example of this.
- The user interacts with the application through an interface (*interface user interaction*, as described in section *5.4.4.2*). The start of the *fire alert*

scenario, where the user selects the skin color of the fireman, is an illustration of this.

It is clear that actions always have a ***subject***[20], which is performing them, and possibly an ***object***[21], undergoing the action. Both subject and object are actors from the scenario.

Additionally, an action may both require and return parameters. However, the only possible input parameters are properties belonging to the actors participating in them. For example, the force with which a fireman chops something, could be dependent on his strength, meaning *strength* would be an input parameter for the *chopping* action. Since the actors participating in the action are known (the *subject* and the *object*), and their properties can be accessed, the required parameters are already available to the action. Thus, there is no need to provide a way for declaring extra input parameters.

Return parameters on the other hand, do need to be declared explicitly. For instance, the action of selecting a skin color returns the chosen color as a return parameter.

Important to note is that these actions are only mentioned in the events, using an intuitive name to describe what they do. The formal specifics of what they mean and how they are executed should be modeled using VR-DeMo *interaction modeling* techniques, as explained in section *5.6*. All that is important from the event's point of view are the actors participating in an action, and optionally, the parameters it returns.

The information above leads to the following notations for an active event name :

```
<action-event-name> ::= <subject> <action>
                      | <subject> <action> <object>
                      | <subject> <action> ( <parameterlist> )
                      | <subject> <action> ( <parameterlist> ) <object>
```

The event displayed in *figure 64* is an example of an active event. Its name is

```
fireman ENTERS_2F b
```

where `<subject> = fireman;` `<action> = ENTERS_2F;` and `<object> = b`. Recall that *b* is the building. This event has no return parameters.

[20] The word 'subject' is used in analogy with the grammatical term.

[21] The word 'object' is used in analogy with the grammatical term.

The remainder of section *5.5.2* provides an overview of the different parts of an active event name.

5.5.2.1. Subject

The subject of an action is the most straightforward part. It is simply an actor – or a set of actors – participating in the scenario. In case of a set, all actors therein perform the action simultaneously. As suggested in section *5.4.4.1*, a practical example of this is a real-time strategy game, where the user gives orders to many identical units at the same time. The subject is denoted as an actor reference (section *5.4.2*).

5.5.2.2. Object

There are several ways to select the actor(s) undergoing the action. The simplest one is, again, to reference one of the scenario's defined actors. Note that the object (or the subject) need not be top-level : it is equally possible to reference some subobject of an actor (e.g. *foamBomb.detonator*). Like the subject, the object can in fact be a set of actors, both top-level (e.g. *stressedWorkers*, returning the set of all stressed workers) or not (e.g. *foamBomb.CILINDER*, returning the set of all of the foam bomb's subobjects of the *CILINDER* concept).

Additionally, it is also possible for the object to be a single actor from a set. To understand this, consider the fireman finds a worker somewhere in the building, and talks to it to escort it out. Note that it doesn't really matter which worker the fireman talks to : he has the same options for each one. However, once he has agreed to escort a certain worker out, the follow-up actions (lead the actor to the door, or to a window and let him climb out using a rope) refer to the same worker that was originally talked to. Once the worker is rescued, another one should be searched for, and the process starts over.

In this case, the scenario should contain an event where the fireman talks to a random member of the *stressedWorkers* set, which he can consequently refer to. To model this, a temporary variable is created, which is bound to the worker in question for as long as needed. Once the worker is rescued, the variable is released, and can be reused for rescuing another worker.

Releasing the variable can only be done by using a post-condition (section *5.7.2.6*). Binding it, however, can be done in two ways : using a post-condition, and by specifying the variable's name as the object of some action. In this latter case, the object should be written as

```
<object> ::= <set-reference> <variable-name>
```

Once this event has been executed, *variable-name* is bound to the correct object in the set until it is released through the post-condition of some later event.

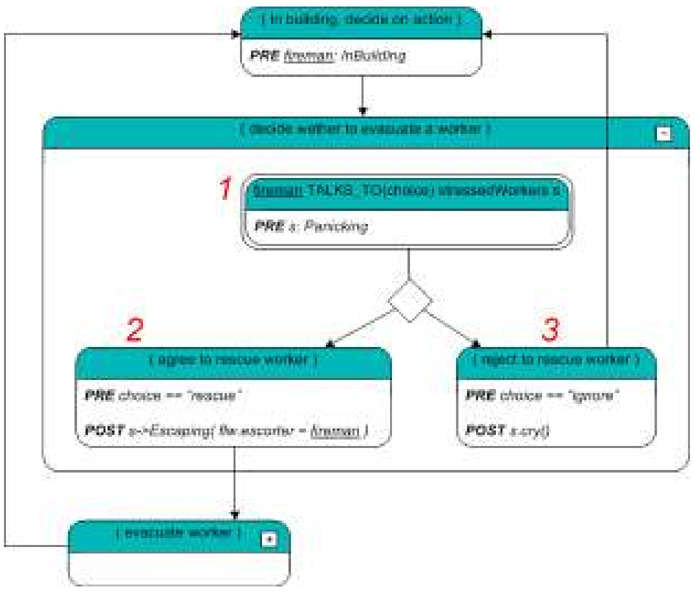

{ in building: decide on action }
PRE fireman: inBuilding
{ decide wether to evacuate a worker }
1
fireman TALKS_TO(choice) stressedWorkers s
PRE s: Panicking
2
{ agree to rescue worker }
PRE choice == "rescue"
POST s->Escaping(file.escorter = fireman)
3
{ reject to rescue worker }
PRE choice == "ignore"
POST s.cry()
{ evacuate worker }

Figure 65 – Fire alert scenario graph fragment : escort decision

Figure 65 shows a fragment of the fire alert scenario graph, modeling the situation described above, where the fireman talks to a worker, and either decides to escort it out or not. The event labeled 1 is an active event, where an actor from the *stressedWorkers* set is bound to the variable *s*. Note that the object notation equals *'stressedWorkers s'*. This variable is consequently used to reference this worker in events 2 and 3 (in their content fields).

5.5.2.3. Action

The action, per convention written in capitals, describes what happens. It is merely a name, nothing more : the actual modeling of the action does not happen in the events, but on different levels of the VR-DeMo approach. As a final step in the construction of a scenario graph, all its actions should be linked to a formal specification (section *5.6*).

For instance, an event modeling a fire being put out by the fireman could be named *'fireman DOUSES fire'*. By reading this name, it is instantly clear what happens in the event. However, the *DOUSES* action might be quite complicated : the closer the fireman is to the fire, the faster it is extinguished; the fireman needs to be facing the fire, he should keep spraying long enough, etc... Luckily, these are issues that the event needs not be worried about.

Note that the scope for the action names is limited to the event itself. Therefore, the designer is free to use whichever names he pleases, to optimally clarify what happens in the event.

The approach of allowing designers to freely specify the name of actions, and later link them to a more formal definition, should allow for more intuitive modeling, and a more understandable diagram (this was inspired by the work discussed in [19]).

5.5.2.4. Return Parameters

If an action should return some value(s), a comma-separated list of return parameter names is added behind the action name, between parentheses. These parameter names provide a link to the lower level at which the actions are modeled : the same parameter names should be used for the return values of the formally defined actions. For example, in event 1 of *figure 65*, the player's decision is assigned to the *choice* parameter. From then on, this name is bound to the player's decision – either *'rescue'* or *'ignore'* – until overwritten.

The namespace of the return parameters is graph-wide : if a different color parameter is defined in a later event, the value from the previous one is overwritten. This is beneficial, since it provides the designer with the opportunity to save the value of a parameter until much later in the scenario graph, or the possibility to reuse names two events after they were first defined.

Return parameters are not typed, they may contain any possible type of value. This should increase both the clarity of the graph and the modeling flexibility, since the same parameter can be reused to contain different types of values.

5.5.3. Passive Events

5.5.3.1. Purpose

Passive events are events that do not actually represent an action being carried out, but are simply used to express extra pre- or post-conditions, facilitate modeling, or merely clarify the graph. In some situations, passive events are required to model some course of actions, while at other times they merely provide the designer with multiple options to model a certain part of the scenario, allowing him to choose the modeling style that he prefers – while still remaining formal.

In certain situations, it is impossible or unpractical to use an active event. For example, suppose that one would like to define an event with a somewhat randomized outcome – like spraying water against a burning door, which may either cause the door to stop burning, or collapse. *Figure 66* shows this example situation.

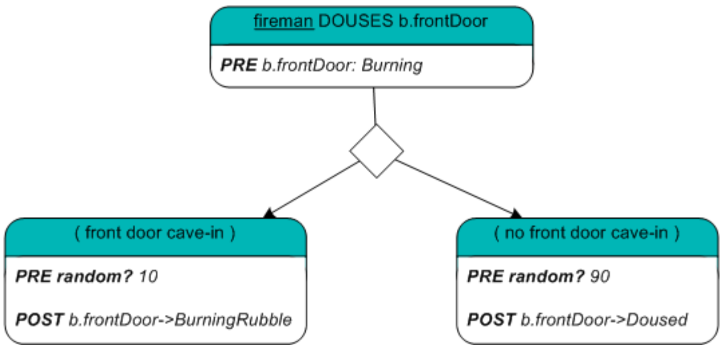


Figure 66 – Non-deterministic event, modeling the possible consequences of attempting to extinguish a burning door

In such a case, it is unfeasible to model the post-conditions within the initial event, since they might vary depending on which outcome is selected. Therefore, a new event is added for each possible result, defining both the probability that the result is chosen, and its post-conditions. Clearly, these resulting events do not represent an action being carried out, so they can not be modeled as active events.

Passive events are modeled in the exact same way as active events, with the exception that their name can be chosen freely. The only requirement is enclosing the name in parentheses, to make the distinction from active events even clearer. Since passive events do not signify an action being performed, there is obviously no need for passive event names to be linked to a formal action definition.

Roughly speaking, three types of passive events can be distinguished :

- ***Nondeterministic events:*** Like the example displayed in *figure 66*. As the name suggest, they are used to model an event with a nondeterministic outcome.
- ***Complex events:*** Events containing a subgraph. They are used to include a higher-level structure in the scenario graph. A detailed explanation is provided in section *5.5.3.2*.
- ***Branching events:*** Branching points in the scenario, usually used for points where a multitude of followup events are available, or where many events come together. These events often have a name prompting the user to do something, like *'select action'*.

Even though passive events can be used for several notably different purposes, they all use the same notation. This is because, while the events are used differently, they all have in common that they do not contain an action, and that they are used to enhance the structure of the graph. Introducing three different notations for these three types of events instills more confusion than it would solve, particularly due to the possibility of multiple uses of the passive event to appear simultaneously (e.g. an event may be a complex and a branching event at the same time).

5.5.3.2. Complex and Reused Events

Aside from conditions, the content field of a passive event may also contain a subgraph. In such a case, the event is said to be a ***complex event***. Events containing a subgraph are automatically labeled with a *minimize / maximize* button in the right corner of the *name field*, allowing the designer to collapse complex events so only the name is shown. Other than that, the notation is identical to regular events. Complex events are useful for clarifying the structure of the diagram, for showing the scenario on a higher level, and for reusing elaborate events inside the same scenario graph, to avoid having to model them twice.

Figure 67 illustrates the concept with a complex event from the fire scenario, modeling the possible sequence of actions taken to enter an office. The reader does not need to be concerned about the unfamiliar specifics of the notation (aside from the events) at this point; they will explained later in this chapter.

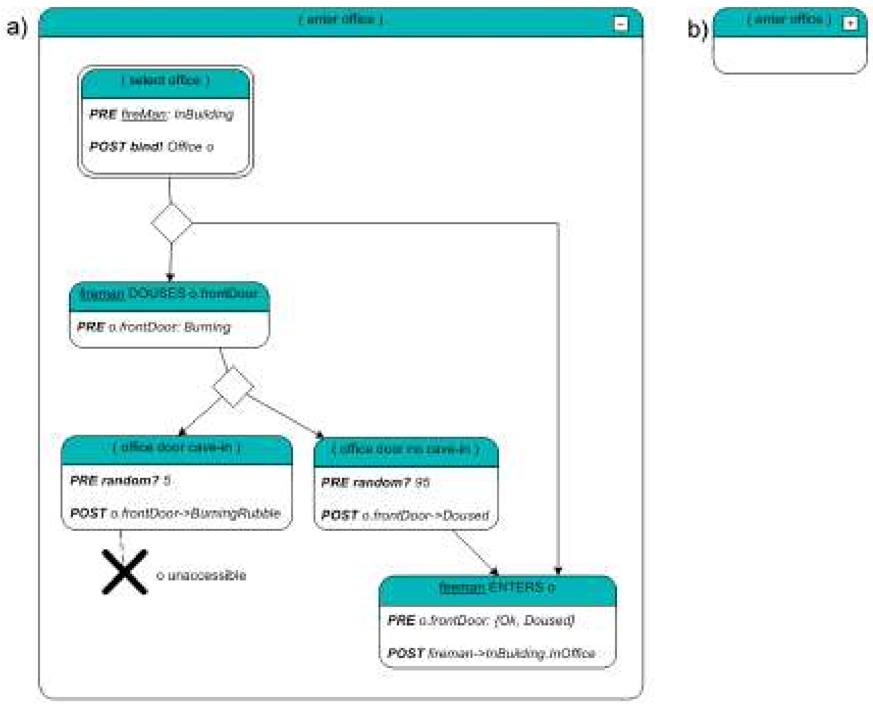


Figure 67 – (a) Full complex event example; (b) Collapsed version of the event

It was mentioned above that complex events can be reused inside the scenario graph. This deserves some extra explanation. Suppose a graph contains a complex sequence of actions which returns in several places during the execution of the scenario. Then, instead of remodeling the whole thing, it would be far easier to simply include a ***reused event*** with the same name, referring to an event defined somewhere else. This is allowed, by using an event with the exact same name, and an arrow symbol in the left corner of the name field. The contents field is left blank. Note that a reused event may be a start event while the original event is not, and vice versa. *Figure 68* shows how to refer to the complex event from *figure 67* by means of a reused event.

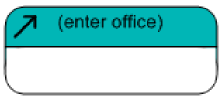


Figure 68 – Notation for reusing the event from *figure 67*

5.6. Actions

5.6.1. Action Types

As described in section *5.5.2*, active events contain some action to be executed by a subject, and possibly some object acted upon, and required return parameters. It was also mentioned that actions are simply written as strings within an event, and that their formal specification should be defined elsewhere. The current section describes the approach taken for this, thereby providing a firmer integration of VR-SDL within the lower level VR-DeMo modeling techniques.

Before looking at how to model things, consider the possible action types based on the subject executing the action :

- ***AI action :*** the subject is an AI-controlled actor, the object (if present) might either be AI controlled or user controlled.
- ***Avatar action :*** the subject is an avatar, controlled by the user, the facultative object is AI controlled.
- ***User action :*** the subject is the user, who interacts with the application on meta-level (e.g., through menus and interfaces). Possible objects might be either AI- or user-controlled. If an object is present, this means the user interacts with it through high-level interaction devices (for example, by clicking on an object using his mouse).

5.6.2. Defining Actions

The contrast between an *AI action* on one hand, and *avatar* and *user actions* on the other hand, is apparent. In the former case, the action is entirely deterministic, as it is executed by the AI, and thus, dependent on program code. Any possible randomness in the execution or outcome of the action can only occur because it is added by the designer. Avatar and user actions, however, are inherently non-deterministic : it is impossible to predict exactly what the user will do, which buttons he will press, and for how long. Therefore, it seems clear that a different modeling approach will have to be taken for action, based on the involvement of the user.

Initially, consider an ***AI action***. Suppose an AI-controlled fireman would douse a flaming door. In this case, the subject is the fireman, the object is the door, and the action is to douse the object. The approach to take for modeling this is quite straightforward : modeling actions performed by some AI object, possibly dependent on another object, is precisely what VR-WISE behaviors (section *2.4*) are used for. Hence, to model an AI action, a new behavior is defined, performed by an actor[22] of the subject's concept, and possibly including a reference actor of the object's concept. In this particular case, a *douse(DOOR door)* behavior would be defined for the *FIREMAN* concept. Roughly, this behavior would model the fireman approaching the fire to a certain distance, facing it, and squirting it with foam for a preset amount of time.

Now suppose that in the dousing action above, the fireman is controlled by the player. In this case, the action is an ***avatar action***, and a behavior is insufficient to model the possible actions taken by the player : while he would take the same *approach – aim – squirt* steps as the AI actor, he might get closer than specified in the behavior, not aim perfectly at the fire, stop squirting foam too early so the fire is reanimated, and so on. Since the focus in this case lies far more on the actions taken by the user – who controls the avatar – the NiMMiT approach (section *2.5*) seems appropriate to model this situation. One would create a NiMMiT diagram with the subject and object as input parameters, and the output parameters specified by the action (if any). However, it is possible that the existing NiMMiT approach will have to be slightly extended to allow for greater compatibility with the VR-WISE *object* and *behavior modeling* techniques, as these might both influence the

[22] The word 'actor' is used in the sense of an actor executing a behavior here, rather than a scenario actor.

action being performed (for one, actor properties might influence the execution of the action, as described in section *5.5.2*).

The final type of action, a ***user action***, is more straightforward to model, since the NiMMiT approach was designed precisely for the purpose of modeling human-computer interaction. Combined another part of VR-DeMo, *VRIXML* [7], even a language for modeling the composition and contents of the interface is available.

5.6.3. Linking Actions to Action Definitions

Now that the possibilities for defining actions are apparent, all that remains is to discuss how to link them to events. In fact, no formal specifications for this exist in VR-SDL. There is no formal notation for linking an action name, as declared in an event, to an action definition. The reason lies in the visual overhead such a notation would entrain. Since actions would have to be uniquely referable, at least one of the following would have to hold :

- **Action names should be unique graph-wide.** This would likely be achieved by adding an id at the end of the action, hampering readability and complicating the modeling, particularly in large graphs.
- **Events should be given a unique id.** While this approach would maintain the clarity of the action names, it would fore an extra id to be included in every event, therefore cluttering up the diagram with meaningless text.
- **The linking should be written down in the event's content field.** This has the same problem as the previous technique, but much more severe. It is particularly unadvisable because – from a scenario viewpoint – how actions are defined does not matter. Therefore, it would be particularly bad modeling to explicitly write the link down in each active event.

The conclusion from the above is that the downsides of forcing a formal link specification are much more significant than the upsides (a quick and easy overview of the lower level design of a scenario's actions). Therefore, the designer is given freedom in how to define the links : in a dedicated tool, each event would likely be given a menu in which to link the action definition, while on paper, a logical approach would be to write a table in combination with the event id technique.

5.7. Conditions

Both active and passive events may contain conditions in their *content field*. There are two kinds of conditions, ***pre-conditions*** and ***post-conditions***: the former are conditions which need to be fulfilled for the event to be able to occur, while the latter describes side-effects of the event.

Care should be taken when trying to interpret the valid conditions in a scenario : they are determined by the path the scenario has taken through the graph. After all, when an event has been executed, its post-conditions are true. Thus, the valid conditions at any point during the execution of the scenario, are all the post-conditions of events which have been executed, except of course the ones that were overwritten by a later event.

This section describes the different types of available pre- and post-conditions. For each type of condition, the notation and a short explanation is provided. Examples of most types of transitions can be found in the scenario graph of the *fire alert* case study (appendix *C*).

5.7.1 Pre-Conditions

Pre-conditions are labeled with a `PRE` keyword. There are several different types of pre-conditions :

- ***Actor state condition***
- ***Actor property condition***
- ***Actor set condition***
- ***Actor exists condition***
- ***Randomness condition***

Each of these conditions types is briefly described below.

5.7.1.1. Actor State Condition

This is the most common type of pre-condition. The basic idea is very simple : it states that an actor – or group of actors – is in a certain state, exactly like in actor graphs (section *4.7.2*). Naturally, actors are referenced as described in the section *5.4.2*. To increase continuity, actor state conditions use roughly the same notation as state conditions in actor graphs :

```
<asc-without-parameters> ::= PRE <actor-ref>:<state>
```

However, when comparing the two, there is an important addition to be contemplated.

Consider the *fire alert* scenario. After a worker *w* has been rescued, he should be in the *Rescued* state. Within this state, the worker has the following behavior defined, indicating he is cheering at his rescuer : *'DO chr = cheer(FIREMAN target)'*. Note that, since the behavior includes pointing at the rescuer and patting him on the shoulder when he is nearby, it has a parameter target specified – otherwise, the worker would not know where to point or pat.

However, when looking exclusively at the actor graph, the object still does not know which object to cheer at – since the *target* parameter is not bound to an object. Therefore, by simply writing the actor state condition as *w:Rescued*, it is unknown which object the worker is cheering at, and thus, who his rescuer is. This is clearly insufficient : instead of stating the worker has been rescued, one should declare by whom he was rescued.

This is denoted by assigning some actor to the *target* parameter. In order to achieve this, the previously described state condition notation is extended with an optional list of parameters, linking the parameters of the state's behaviors to actors from the scenario (in this particular case, *fireman*). Since a state will rarely contain more than one behavior, and even less frequently more than one behavior with parameters, this notation will not cause extensive overhead.

Care should be taken to specify exactly which parameters a value is assigned to. Thankfully, all the tools required for such an endeavor are already available :

- *State* and *transition behaviors* are named, so they can be referenced - using a scope delimiter if needed (section *4.8.2.2*).
- For accessing a certain parameter in a behavior, the dot-operator is used.
- An actor is assigned to a parameter by using an equals sign.

This leads to an extension of the notation, as shown below :

```
<asc-with-parameters> ::= PRE <actor-ref>:<state> (
                              <behav-parameter-list> )

<behav-parameter> ::= <behav-ref>.<parameter-name> = <actor-ref>
```

Using this notation, the full fire example condition would be specified as follows : *PRE w:Rescued(chr.target=fireman)*.

Note that the majority of the states in actor graphs will not contain parameterized behaviors, in which case the parameter list is not needed, and the more compact condition notation from the start of this section is used.

5.7.1.2. Actor Property Condition

An actor property condition signifies that a certain actor property is equal to, greater than or smaller than some value. Again, the dot-operator is used to access a property. Additionally, actor property conditions can be used on all elements of a set. Rather than using a forced, shorter notation using only the set name, a longer, but more intuitive notation is used, employing a `foreach` keyword.

Notation for a single actor :

```
<apc-single-actor> ::= PRE <actor-ref>.<property> <comparator> <value>
```

Notation for an actor set :

```
<apc-actor-set> ::= PRE (foreach <name> in <set-ref>) <name>.<property>
                        <comparator> <value>
```

5.7.1.3. Actor Set Condition

An actor set condition performs a test related to an actor set (or subgroup). Currently, two actor set conditions are defined : testing whether a set is empty, and testing whether some actor is in a set.
In this type of condition, and the next two, a special keyword is used. These keywords are always succeeded by a question mark, to clarify the fact that they are pre-condition keywords.

Notations :

```
<actor-set-condition> ::= PRE empty? <set-ref>
                        | PRE in? <actor-ref> <set-ref>
```

5.7.1.4. Actor Exists Condition

Tests whether a certain actor exists. Non-initialized actors return false here. This condition provides a way to check whether an actor has been created or deleted. It is in a sense the single actor equivalent of an actor set condition.

Notation :

```
<actor-exists-condition> ::= PRE exists? <actor-ref>
```

5.7.1.5. Randomness Condition

Defines the probability of this event being chosen after a branching transition (section *5.8*). If a single event at the end of a branching transition contains a randomness condition, so does the rest, and the sum of their probabilities should equal 1. The probability value can be any expression, for example, some object property might be used to determine it.

Notation :

```
<randomness-condition> ::= PRE random? <probability-value>
```

5.7.2. Post-Conditions

Post-conditions express side-effects of the event being executed. Hence, post-conditions always indicate some change, while pre-conditions imply a continuity from some previous event. Unsurprisingly, post conditions are labeled with a `POST` keyword. The following types of post-conditions are distinguished :

- ***State change condition***
- ***Property assignment condition***
- ***Actor creation / destruction condition***
- ***Subgroup manipulation condition***
- ***Behavior execution condition***
- ***Bind / release condition***
- ***Avatar change condition***

Many of these are counterparts to an equivalent pre-condition. They are explained below.

5.7.2.1. State Change Condition

A state change condition indicates a change of state in a certain actor, or actor set. It is specified in the same way as an actor state pre-condition (section *5.7.1.1*), with a two alterations :

- Instead of the colon notation, an arrow -> is used, to indicate the actor changes state, rather than simply being in one.
- If a transition behavior is present, it is possible that this behavior will contain parameters (e.g. a '*DO pt = point(FIREMAN target)*' behavior). Therefore, it is required to assign some actor to these parameters as well. The notation is entirely analogous to that of state behavior parameters.

Notation without parameters :

```
<ssc-without-parameters> ::= POST <actor-ref> -> <state>
```

Full notation :

```
<ssc-with-parameters> ::= POST <actor-ref> -> <state> (
                               <behav-parameter-list> )

<behav-parameter> ::= <behav-ref>.<parameter-name> = <actor-ref>
```

5.7.2.2. Property Assignment Condition

The equivalent of the actor property pre-condition, property assignment conditions assign a new value to a property of an actor. Like the actor property pre-condition, property assignment conditions can be specified on a set, using a foreach notation.

Notation for a single actor :

```
<pac-single-actor> ::= POST <prop-ref> = <value>
```

Notation for an actor set :

```
<pac-actor-set> ::= POST ( foreach <actor-name> in <set-ref> )
                         <prop-ref> = <value>
```

5.7.2.3. Actor Creation / Destruction Condition

Recall from section *5.4.1* that it is possible to define a non-initialized actor, which is available for initialization by the scenario. This is achieved by using an actor creation condition. Furthermore, an actor graph (and thus, a role) may serve as the parent graph for one or more child graphs (section *4.11*). In such a case, the actor's role may be defined as the parent graph, while the actual object linked to it is an instance of one of the child graphs. To specify this, the type of child which is created is written in the creation condition.

Alternatively, existing actors can be destroyed by using an actor destruction condition. Destruction conditions can also be defined on an actor set, prompting the destruction of all actors in the set.

In this condition and the following ones, special keywords are used, delimited by an exclamation mark, to indicate they are causing a change in the VE.

Notations :

```
<actor-condition> ::= POST create! <actor-name> ( <x>, <y>, <z> )
                    | POST create! <actor-name> as <actor-role-child> (
                           <x>, <y>, <z> )
                    | POST destroy! <actor-name>
                    | POST destroy! <set-name>
```

5.7.2.4. Subgroup Manipulation Condition

Subgroup manipulation conditions add an actor to a set, delete it from one, clear a set, or copy its elements into a different set.

Notations :

```
<subgroup-condition> ::= POST add! <actor-ref> <set-ref>
                       | POST remove! <actor-ref> <set-ref>
                       | POST clear! <set-ref>
                       | POST copy! <set-ref> to <set-ref>
```

5.7.2.5. Behavior Execution Condition

A behavior execution condition prompts the execution of a certain behavior by an actor. It is much simpler than the behavior specification within states, however : the behavior is executed exactly once, and it needs no special naming.

Notation :

```
<behavior-execution-condition> ::= POST <actor-ref>.<behav-ref>(
                                        <behav-parameter-list> )
```

5.7.2.6. Bind / Release Condition

A bind condition binds a certain object from a set to a variable name. It remains bound until it is released by a release condition. Note that objects can also be bound to names by using a special notation structure for the *object* in an active event name. For a full explanation, refer to section *5.5.2.2*.

Notation :

```
<bind-release-condition> ::= POST bind! <actor-name> <actor-ref>
                           | POST release! <actor-name>
```

5.7.2.7. Avatar Change Condition

An avatar change condition defines a change in the user's avatar. In other words, the user will stop controlling a certain actor in the world, and start controlling a different one. This was described in section *5.4.4.1*.

Notation :

```
<avatar-change-condition> ::= POST setAvatar! <actor-name>
```

5.8. Transitions

A transition in a scenario graph connects an event to its possible follow-up events. Unlike actor graph transitions, transitions in a scenario graph are never labeled with a trigger : it is the subject of an event which decides whether the event gets executed (by performing the appropriate action).

Like in actor graphs (section *4.10*), both one- and two-way transitions are available. They use the same notation : a full line with an arrowhead on one or both sides, respectively. Unlike in actor graphs, each event can only have a single transition leaving it. If the event has more than one follow-up, this transition branches towards each of them. The branching point is drawn as a diamond, a notation introduced for a better visualization of the choices in flow of the scenario. *Figure 69* shows the different transition types.

Be advised that a branching transition in a scenario graph is interpreted differently from branching transitions in actor graphs : in scenarios, they connect one event to possible follow-ups, while in actor graphs, they connect a number of states that can be randomly changed between.

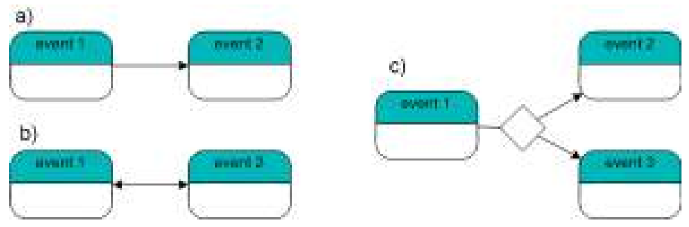


Figure 69 – (a) One-way transition; (b) Two-way transition; (c) Branching transition

One final remark remains : complex events (section *5.5.3.2*) may contain transitions connected to some lower level events, instead of the complex event itself. When such a complex event is collapsed, these transitions are shown as connected to the top-level event, to preserve the structure of the graph.

5.9. Terminals

Due to the inevitably complex nature of scenario graphs, it is easy to lose track of certain important key points in the scenario. Therefore, an extra construct is introduced to denote 'points of no return' in the scenario, simply as a visual aid to the designer or a reader of the diagram : ***terminals***.

A terminal is a black cross, connected to an event with a dashed line. It is always labeled with some text, describing what bridges have been blown up. Since this is an informal construct, solely used for increased readability, there are no special rules for writing the text. It is merely a natural language sentence.

An example from the fire scenario is the collapse of the fire escape : once this has happened, the fire escape is forever inaccessible, since the fireman has no means to repair it. Therefore, the event describing the collapse of the fire escape might have a terminal connected to it, as shown in *figure 70*, to clarify this milestone in the scenario.

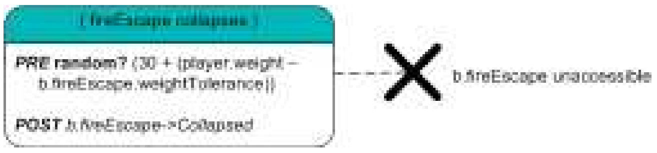

{ fireEscape collapses }
PRE random? (30 + (player.weight – b.fireEscape.weightTolerance))
POST b.fireEscape->Collapsed
b.fireEscape unaccessible

Figure 70 – Terminal signifying the collapse of the fire escape

5.10. Conclusion

In this chapter, the notation and modeling concerns for scenario graphs were discussed. With the information presented in this chapter and the previous one, all that is necessary to model a scenario using VR-SDL has been put forward.

Before creating a scenario graph, its actors should be formally defined. A textual notation is used for this, specifying the names and types of all actors involved. Additionally, actor sets and subgroups can be defined, representing a group of actors with similar behavior in the scenario.

The scenario graph itself is a type of flowchart, describing the possible execution flows for the scenario. Scenario graphs may contain the following elements :

- **Events :** Something that happens in the scenario. Two types of events exist : active events, representing an action taken by certain actors, and passive events, used for modeling extra conditions, important branching points in the scenario, and higher level or non-deterministic events.
- **Actions :** The actions modeled in active events. They are formally specified on a lower level, either by *behavior modeling* or by *interaction modeling*.

- **Conditions :** Pre-conditions describe requirements for an event to be executable, and post-conditions express side effects of an event's occurrence.
- **Transitions :** Connect events to their possible follow-ups, thus specifying the possible flow of the scenario.

Chapter 6

Conclusion and Further Work

6.1. Conceptual Modeling of VR using VR-DeMo

In this thesis, **VR-SDL**, a graphical scenario modeling language for virtual environments, was presented.

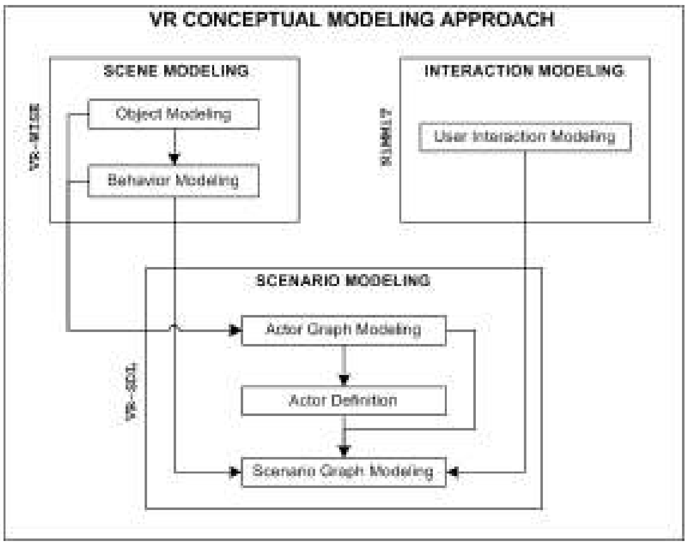


Figure 71 – An overview of modeling a VR application using VR-DeMo

6.1.1. Scene and Interaction Modeling

VR-SDL is a part of the overall **VR-DeMo** approach (*figure 71*), a methodology developed for the conceptual modeling of virtual reality applications. Such a methodology has the potential to greatly reduce the time and costs involved in creating a virtual environment, since it eliminates the designer – implementer overhead, and because conceptual diagrams may be used as a basis for the generation of source code. In its current form, VR-DeMo provides support for modeling the scene of a virtual environment and the behavior of the objects therein. Furthermore, it also allows modeling user interaction with such a virtual environment, through interfaces and menus.

Modeling the scene of a VE together with the behavior of its objects is done using **VR-WISE**, an approach consisting of two steps (as shown in *figure 71*). Primarily, there is an *object modeling* phase, where the static objects populating the world are modeled. These objects may be given a graphical representation and some properties. For example, in an urban VE, objects modeled might include an office building, the components of that building (offices, doors, windows...), the workers populating it, and some additional types of people, such as a fireman. A fireman could have properties such as strength or face color. Additionally, there is a *behavior modeling* phase : after the objects have been designed, they may be given behavior to increase the realism of the VE. In the office building, for instance, workers may be talking to one another, walking towards a vending machine when they get hungry, or typing a report on their computer.

Being able to model such a VE is very nice, but it has very little utility if the user can not interact with it. Therefore, VR-DeMo contains an *interaction modeling* approach called **NiMMiT**. Using NiMMiT, one may for example model the actions the user would have to take to change the skin color of some fireman.

6.1.2. Scenario Modeling

Even with the addition of interaction modeling, the virtual environment described in section *6.1.2* still lacks a purpose. One could imagine it being used in a shooting game, where the player would have to infiltrate the office building and eliminate some terrorists occupying it. A different game could have the building set aflame, and the player controlling a fireman, tasked with rescuing as many workers as possible. Alternatively, the VE may be employed for a room furnishing application, where the user can redecorate his own office, and walk around in it with a worker-avatar. Another use might be that of a 3D chat room, where multiple users inhabit the world simultaneously, each controlling their own avatar, talking to their friends online. Though all these applications use the same underlying VE, they are clearly very distinct. Using only VR-WISE and NiMMiT, it is impossible to design this with VR-DeMo.

Therefore, **VR-SDL** is introduced, a graphical language for modeling scenarios for VEs. A scenario is a sequence of events happening in the world, possibly influencing its objects. Any of the illustrated uses above could be seen as a single scenario, describing the possible occurrences in the VE. For example, if the application were a shooting game, the scenario would include the possibility of some worker being shot, while this would be impossible in any of the other applications.

A dual approach is used for modeling scenarios with VR-SDL : first, the objects participating in the scenario (called *actors*) are modeled using **actor graphs**, and after that, the scenario itself is designed as a **scenario graph**.

6.1.2.1. Actor Graphs

Actor graphs, a special kind of finite state machines, describe the way in which objects may evolve during the course of a scenario. Consider using the urban VE for the *fire alert* game mentioned above (a complete description of this scenario was presented earlier in this thesis, in section *5.2*). In this scenario, many actors (such as doors, windows, offices...) may be either on fire or not on fire. This information is important to the course of the scenario : a door which is on fire can be extinguished, but a door which is not burning obviously can not.

Actor graphs contain many facilities to express a complicated evolution during a scenario, such as :

- Different behaviors may be executed in different states (i.e. a burning door displays a flaming animation, while a non-burning door does not).
- Property values could be changed (i.e. the fireman may be given a different skin color at the start of the scenario).
- The visual appearance of an object may be altered (i.e. a burning door looks different from an extinguished door).

Many of these expressions are closely tied to the constructs modeled using the lower level VR-WISE approach. A full specification of the actor graphs of the *fire alert* scenario, demonstrating the notation, may be found in appendix *C*.

6.1.2.2 Scenario Graphs

When the actor graphs have been created, all that remains is the scenario itself. Scenarios are modeled using scenario graphs, a special type of flowchart. Before designing this graph, the actors in a scenario need to be formally defined. In an actor definition, a scenario actor is given a name, an actor graph, and it is possibly linked to some object from the VE. For example, while there may be multiple firemen present in the VE, the fire alert scenario only requires one. Through the actor definition, one of the fireman objects is selected as the actor in the scenario. Furthermore, actor definitions also specify which actor (if any) is controlled by the user.

An actual scenario graph describes a web of **events** occurring in the world (i.e. the fireman douses a door), and their **post-conditions** : these are the effects that executing the event has on the remaining actors (the aforementioned door would change from being on fire, to being doused). Events might also have certain **pre-conditions** that should be fulfilled before they are able to occur. Through **transitions**, scenario graphs define the way in which events may follow up on each other, thus defining the possible flow of the scenario. For example, a fireman can only enter a room after having extinguished its door, and once he is inside a room, he can either talk to some worker inside it, douse the room's window, or go back outside again.

The full scenario graph for the fire alert scenario is included in appendix *C*.

6.2. Further Work

While VR-SDL is undoubtedly an interesting first step towards a finalized scenario modeling language, it would be quite pretentious to state that it is perfect in its current form. The results of the research performed for this thesis could be labeled 'VR-SDL 1.0', and like any modeling, programming or scripting language, this initial version should evolve as the needs of its users dictate. Hence, the most important work to be performed in the future is experimentation with the VR-SDL approach.

Throughout the thesis, it was often stated that the intuitiveness of the notation was an important design guideline while creating VR-SDL. However, during the creation of the approach, all one can do is to attempt to achieve intuitiveness, based on precedents set by popular existing languages such as UML. Whether VR-SDL is in fact intuitive or not, should be examined through user experiments. Such experiments would comprise of asking a group of volunteers to familiarize themselves with the notation, instruct them to model a few simple scenarios, and record their feedback. In order to obtain a valid sample of VR-SDL's target audience, care should be taken to include people with different backgrounds, ranging from programmers to domain experts with lacking computer skills. Through such experiments, it would be interesting to investigate – among other things – whether the adoption of Java naming conventions does not make the notation harder to grasp for people unfamiliar with Java (or a language with similar conventions). Additionally, feedback would be provided about the intuitiveness of certain constructs or construct names.

Additional case studies should be performed for examining the completeness of the VR-SDL notation. This could be evaluated by collecting a few scenarios (possibly rather complex) from existing applications. Next, a VR-SDL expert would try to model these scenarios, to track down possible limitations in the approach. The most probable extensions include new pre- or post-conditions for the events in scenario graphs.

One such gap in the present state of the approach, is support for scenarios with multiple users. This is currently not supported by VR-SDL, and should be added in the future. It would likely involve user definitions, next to actor definitions, and allow to link actors to a user as well as an instance. Some other possible extensions include the definition of common scenario patterns (similar to design patterns in programming [12]), or the introduction of scenarios that can change at run-time (for example, under the influence of a designer).

Furthermore, the suitability of NiMMiT for modeling avatar interaction is open for debate. There is a possibility that the current NiMMiT approach would have to be extended to provide a tighter link with the output of the VR-WISE object and behavior modeling phases.

Finally, the ability to generate source code for scenarios deserves looking into. Currently, the diagrams created by VR-WISE are already used to generate source code for the VE, and it would be interesting to see if the same can be done with VR-SDL diagrams – and to what extent.

Bibliography

1. Badler, N., et al., A Parameterized Action Representation for Virtual Human Agents, in *Embodied Conversational Agents*. 2001, MIT Press: Cambridge, MA, USA. p. 256-284.

2. Bartle, R.A., Designing Virtual Worlds. 2003, Indianapolis: New Riders.

3. Bartley, C.R., A Visual Language for Composable Simulation Scenarios. 2003, Air Force Institute of Technology: Ohio. p. 146.

4. Beckhaus, S., et al. alVRed - Methods and Tools for Storytelling in Virtual Environments. in *Internationale Statustagung Virtuelle und Erweiterte Realität*. 2004. Leipzig, Germany.

5. Bille, W., O. De Troyer, and R. Romero, OntoWorld: a Tool to Generate Virtual Worlds from Conceptual Specifications, Vrije Universiteit Brussel: Brussels.

6. Bille, W., et al. Intelligent Modelling of Virtual Worlds Using Domain Ontologies. in *Workshop of Intelligent Computing*. 2004. Mexico City, Mexico.

7. Cuppens, E., C. Raymaekers, and K. Coninx. VRIXML : A USer Interface Description Language for Virtual Environments. in *ACM AVT'2004 Workshop: Developing User Interfaces with XML: Advances on User Interface Description Languages*. 2004.

8. De Troyer, O., et al. On Generating Virtual Worlds from Domain Ontologies. in *The 9th International Conference on Multi-Media Modeling*. 2003. Taipei, Taiwan.

9. Dries, J., Adaptive Storylines in Dynamic Gaming Environments. 2004, Vrije Universiteit Brussel: Brussel. p. 75.

10. Fowler, M., UML distilled. A Brief Guide to to Standard Object Modeling Language. 3 ed. 2003, Boston: Pearson Education. 175.

11. Funcom. Funcom and Massive Inc. to implement revolutionary game ad deal. 2005 [cited 2007 30/4]; Available from: *http://www.anarchy-online.com/content/news/articles/8526L*.

12. Gamma, E., et al., Design Patterns: Elements of Reusable Object-Oriented Software. 1995, Indianapolis: Pearson Education.

13. Grützmacher, B., R. Wages, and G. Trogemann. An Authoring System for Non-Linear VR Scenarios. in *The 9th International Conference on Virtual Systems and Multimedia*. 2003. Montreal, Canada.

14. Hargens, T. New Virtual Reality Surgery Simulator Hones Surgeons' Skills, Improves Patient Safety. 2005 20/6 [cited 2007 29/4]; Available from: *http://www.ohsu.edu/ohsuedu/newspub/releases/062005virtual.cfm*.

15. Houlette, R., D. Fu, and D. Ross. Towards an AI Behavior Toolkit for Games. in *AAAI 2001 Spring Symposium on AI and Interactive Entertainment*. 2001. Stanford University, California.

16. IKEA. KeukenPlanner. 2007 [cited 2007 May 27th]; Available from: *http://www.ikea.com/ms/nl_BE/complete_kitchen_guide/planner_tool/index.html*.

17. Ishida, T., Q: A Scenario Description Language for Interactive Agents, in *Computer*. 2002. p. 42-47.

18. Noy, N.F. and D.L. McGuinness. Ontology Development 101: A Guide to Creating Your First Ontology. 2000 [cited 2007 June 1st]; Available from: *http://www-ksl.stanford.edu/people/dlm/papers/ontology101/ontology101-noy-mcguinness.html*.

19. Ohnishi, A. and C. Potts. Grounding Scenarios in Frame-Based Action Semantics. in *Requirements Engineering: Foundation of Software Quality (REFSQ'01)*. 2001. Interlaken, Switzerland.

20. Pellens, B., et al. VR-WISE: A Conceptual Modeling Approach for Virtual Environments. in *Methods and Tools for Virtual Reality Workshop (MeTo-VR 2005)*. 2005. Gent, Belgium.

21. Pellens, B., et al., Conceptual Modeling of Behavior in a Virtual Environment. International Journal of Product and Development, 2007(Special Issue).

22. Pellens, B., et al., Virtual Reality: Conceptual Descriptions and Models for the Realization of Virtual Environments. A description of a set of high-level modeling concepts for specifying simple and complex objects and behaviour in Virtual Worlds, Vrije Universiteit Brussel: Brussel.

23. Pellens, B., F. Kleinermann, and O. De Troyer, Virtual Reality: Conceptual Descriptions and Models for the Realization of Virtual Environments. A Description of a Set of High-Level Modelling Concepts for specifying more Advanced Behaviour in Virtual Environments, Vrije Universiteit Brussel: Brussel.

24. Perlin, K. and A. Goldberg. Improv: A System for Scripting Interactive Actors in Virtual Worlds. in *The 23th Annual Conference on Computer Graphics and Interactive Techniques*. 1996. New Orleans, LA, USA: ACM Press.

25. Rabin, S., AI Game Programming Wisdom. 2002, Boston: Charles River Media.

26. Rabin, S., AI Game Programming Wisdom 2. 2003, Boston: Charles River Media.

27. SCEE. Go Home with PS3. 2007 08/03 [cited 2007 30/4]; Available from: *http://uk.playstation.com/games-media/news/articles/detail/item55853/Go-Home-with-PS3/*.

28. Slaughter, L. Flight Simulators. [cited 2007 29/4]; Available from: *http://www.ebaft.com/index1.htm*.

29. Vanacken, D., et al. NiMMiT: A Notation for Modeling Multimodal Interaction Techniques. in *First International Conference on Computer Graphics Theory and Applications*. 2006. Setúbal, Portugal.

30. Vargas, J.A. Virtual Reality Prepares Soldiers for Real War. 2006 14/2 [cited 2007 29/4]; Available from: *http://www.washingtonpost.com/wp-dyn/content/article/2006/02/13/AR2006021302437.html*.

31. Wages, R., Non-Linear Scriptwriting - A Graph-Based Authoring Tool for Interactive Scenarios. 2004: Stuttgart, Germany.

32. Wages, R., B. Grützmacher, and S. Conrad. Learning from the Movie Industry: Adapting Production Processes for Storytelling in VR. in *Technologies for Interactive Digital Storytelling and Entertainment: Second International Conference, TIDSE 2004*. 2004. Darmstadt, Germany.

33. Wages, R., B. Grützmacher, and G. Trogemann. A Formalism and a Tool for Diverging Requirements in VR Scenario Modeling. in *The 13th International Conference on Artificial Reality and Telexistence*. 2003. Keio University, Tokyo, Japan.

34. Weber, T. Games industry enters a new level. 2007 11/4 [cited 2007 30/4]; Available from: *http://news.bbc.co.uk/2/hi/business/6523565.stm*.

35. Willemsen, P.J., Behavior and Scenario Modeling for Real-Time Virtual Environments. 2000, University of Iowa: Iowa. p. 172.

36. WISE. Web & Information Systems Engineering. 2004 [cited 2007 30/4]; Available from: *http://vr-wise.vub.ac.be/default.htm*.

Appendix A

BNF Specifications

A.1. Actor Graph BNF Specification

```
//Concept-related syntax

<subobject-ref> ::= <role-name>
                  | <concept-name>
                  | <actor-graph-name>
                  | <subobject-ref>.<subobject-ref>

<subobject-link> ::= SUB <role-name> is <actor-graph-name>
                   | SUB <concept-name> is <actor-graph-name>

<prop-ref> ::= <property-name>
             | <subobject-ref>.<property-name>

<property-definition> ::= PROP <property-name>
                        | PROP <property-name> = <value>
                        | PROP <property-name> { <enum-list> }
                        | PROP <property-name> { <enum-list> } = <value>

<enum-list> ::= <enum-value>
              | <enum-value>, <enum-list>



//State-related syntax

<qualification> ::= <property-qualification>
                  | <state-qualification>

<property-qualification> ::= QLF <prop-ref> = <value>
                           | QLF <prop-ref> = [ <num-value>, <num-value> ]

<state-qualification> ::= QLF <subobject-ref> : <state-ref>
                        | QLF <subobject-ref> : { <state-list> }

<state-ref> ::= <state-name>
              | <state-ref>.<state-name>

<state-list> ::= <state-ref>
               | <state-ref>, <state-list>

<do-behavior> ::= DO <behav-name> = <behav-def-ctr>
                | DO <behav-name> = [ <behav-list> ]

<behav-list> ::= <behav-def-ctr>
               | <behav-def-ctr>, <behav-list>

<behav-def-ctr> ::= <behav-definition>
                  | <behav-definition> <execution-counter>

<behav-definition> ::= <behav-specification> ()
                     | <behav-specification> ( <actor-list> )

<execution-counter> ::= * Integer
                      | * ∞

<actor-list> ::= <behav-actor-param>
               | <behav-actor-param>, <actor-list>

<behav-actor-param> ::= <concept-name> <behav-actor-name>

<cont-behavior> ::= CONT <behav-ref>

<behav-ref> ::= <behav-name>
              | <state-ref> :: <behav-name>

<visual> ::= VISUAL <graphics-name>

<overwriting-state-name> ::= <state-name> : <state-name>



//Transition-related syntax
```

```
<transition-trigger> ::= TRIG <behavior-finish-trigger>
                       | TRIG <property-qualification-trigger>
                       | TRIG <state-qualification-trigger>
                       | TRIG <elapsed-time-trigger>

<behavior-finish-trigger> ::= atBehaviorFinish( <behav-ref> )

<property-qualification-trigger> ::= <prop-ref> <comparison-operator> <num-value>

<state-qualification-trigger> ::= <subobject-ref> : <state-ref>

<elapsed-time-trigger> ::= onElapsedTime( <num-value> )
```

//General syntax

```
<role-name> ::= <name>
<concept-name> ::= <name>
<actor-graph-name> ::= <name>
<property-name> ::= <name>
<state-name> ::= <name>
<behav-name> ::= <name>
<behav-actor-name> ::= <name>
<graphics-name> ::= <name>

<name> ::= Identifier

<enum-value> ::= String | Decimal | Integer

<value> ::= String | <num-value>

<num-value> ::= <number>
              | <sign-operator> <value>
              | <value> <arithmetic-operator> <value>
              | ( <value> )

<number> ::= Decimal | Integer | <prop-ref>

<arithmetic-operator> ::= + | - | * | / | div | mod

<sign-operator> ::= + | -

<comparison-operator> ::= < | > | == | <= | >= | !=
```

A.2. Scenario Graph BNF Specification

```
//Actor-related syntax

<actor-definition> ::= <single-actor-definition>
                     | <actor-set-definition>

<single-actor-definition> ::= ACTOR <actor-name> role <actor-role>
                            | ACTOR <actor-name> role <actor-role> playedBy <instance-ref>

<actor-set-definition> ::= ACTORS <set-name> role <actor-role>[]
                         | ACTORS <set-name> role <actor-role>[ Integer ] playedBy
                           [ <instance-list> ]

<instance-list> ::= <instance-ref>
                  | <instance-ref>, <instance-list>

<instance-ref> ::= <instance-name>
                 | <instance-name>.<instance-ref>

<actor-role> ::= Identifier

<actor-ref> ::= <actor-name>
              | <actor-ref>.<actor-name>

<set-ref> ::= <set-name>
            | <actor-ref>.<concept-name>
            | <actor-ref>.<actor-role>


//Event-related syntax

<action-event-name> ::= <subject> <action>
                      | <subject> <action> <object>
                      | <subject> <action> ( <parameter-list> )
                      | <subject> <action> ( <parameter-list> ) <object>

<subject> ::= <actor-ref>

<action> ::= Text

<object> ::= <actor-ref>
           | <set-ref> <actor-name>

<parameter-list> ::= <parameter-name>
                   | <parameter-name>, <parameter-list>

<structure-event-name> ::= ( Text )


//Condition-related syntax

<pre-condition> ::= PRE <pre-condition-single>
                  | PRE <pre-condition-list>

<pre-condition-list> ::= ( <pre-condition-list> )
                       | <pre-condition-or-list>
                       | <pre-condition-and-list>

<pre-condition-or-list> ::= <pre-condition-single>
                          | ( <pre-condition-and-list> )
                          | ( <pre-condition-and-list> ) or <pre-condition-or-list>

<pre-condition-and-list> ::= <pre-condition-single>
                           | ( <pre-condition-or-list> )
                           | ( <pre-condition-or-list> ) and <pre-condition-and-list>

<pre-condition-single> ::= ( <pre-condition-single> )
                         | not <pre-condition-single>
                         | <actor-state-condition>
                         | <actor-property-condition>
                         | <actor-set-condition>
                         | <actor-exists-condition>
```

```
                        | <randomness-condition>

<actor-state-condition> ::= <asc-without-parameters>
                          | <asc-with-parameters>

<asc-without-parameters> ::= <multi-actor-ref> : <state-ref>
<asc-with-parameters> ::= <asc-without-parameters> ( <behav-parameter-list> )

<multi-actor-ref> ::= <actor-ref>
                    | <set-ref>

<state-ref> ::= <state-name>
              | <state-ref>.<state-name>

<behav-parameter-list> ::= <behav-parameter>
                         | <behav-parameter>, <behav-parameter-list>

<behav-parameter> ::= <behav-ref>.<parameter-name> = <actor-ref>

<behav-ref> ::= <behav-name>
              | <state-ref>::<behav-name>

<actor-property-condition> ::= <apc-single-actor>
                             | <apc-actor-set>

<apc-single-actor> ::= <prop-ref> <comparison-operator> <value>
<apc-actor-set> ::= ( foreach <actor-name> in <set-ref> ) <apc-single-actor>

<prop-ref> ::= <actor-ref>.<property-name>

<actor-set-condition> ::= empty? <set-ref>
                        | in? <actor-ref> <set-ref>

<actor-exists-condition> ::= exists? <actor-ref>

<randomness-condition> :: random? <value>


<post-condition> ::= POST <state-change-condition>
                   | POST <property-assignment-condition>
                   | POST <actor-manipulation-condition>
                   | POST <subgroup-manipulation-condition>
                   | POST <behavior-execution-condition>
                   | POST <bind-release-condition>
                   | POST <avatar-change-condition>

<state-change-condition> ::= <scc-without-parameters>
                           | <scc-with-parameters>

<ssc-without-parameters> ::= <multi-actor-ref> -> <state-ref>
<ssc-with-parameters> ::= <ssc-without-parameters> ( <behav-parameter-list> )

<property-assignment-condition> ::= <pac-single-actor>
                                  | <pac-actor-set>

<pac-single-actor> ::= <prop-ref> = <value>
<pac-actor-set> ::= ( foreach <actor-name> in <set-ref> ) <pac-single-actor>

<actor-manipulation-condition> ::= <actor-creation-condition>
                                 | <typed-actor-creation-condition>
                                 | <actor-destruction-condition>
                                 | <set-destruction-condition>

<actor-creation-condition> ::= create! <actor-name> ( <position> )
<typed-actor-creation-condition> ::= create! <actor-name> as <actor-type> ( <position> )
<actor-destruction-condition> ::= destroy! <actor-name>
<set-destruction-condition> ::= destroy! <set-name>

<position> ::= Integer, Integer, Integer

<subgroup-manipulation-condition> ::= <subgroup-add-condition>
                                    | <subgroup-remove-condition>
                                    | <subgroup-clear-condition>
                                    | <subgroup-copy-condition>

<subgroup-add-condition> ::= add! <actor-ref> <set-ref>
<subgroup-remove-condition> ::= remove! <actor-ref> <set-ref>
```

```
<subgroup-clear-condition> ::= clear! <set-ref>
<subgroup-copy-condition> ::= copy! <set-ref> to <set-ref>

<behavior-execution-condition> ::= <actor-ref>.<behav-ref>( <behav-parameter-list> )

<bind-release-condition> ::= <bind-condition>
                           | <release-condition>

<bind-condition> ::= bind! <actor-name> <actor-ref>
<release-condition> ::= release! <actor-name>

<avatar-change-condition> ::= setAvatar! <actor-name>
```

//General syntax

```
<actor-name> ::= <name>
<instance-name> ::= <name>
<set-name> ::= <name>
<concept-name> ::= <name>
<state-name> ::= <name>
<parameter-name> ::= <name>
<behav-name> ::= <name>
<property-name> ::= <name>

<name> ::= Identifier

<value> ::= <number>
          | <sign-operator> <value>
          | <value> <arithmetic-operator> <value>
          | ( <value> )

<number> ::= Decimal | Integer | <prop-ref>

<arithmetic-operator> ::= + | - | * | / | div | mod

<sign-operator> ::= + | -

<comparison-operator> ::= < | > | == | <= | >= | !=
```

Appendix B

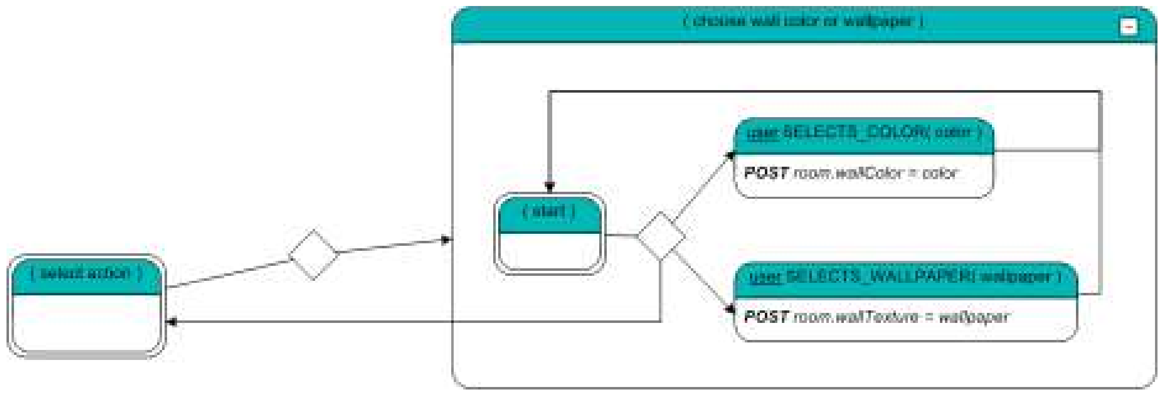
Room Furnishing Example

B.1. Actor Definitions

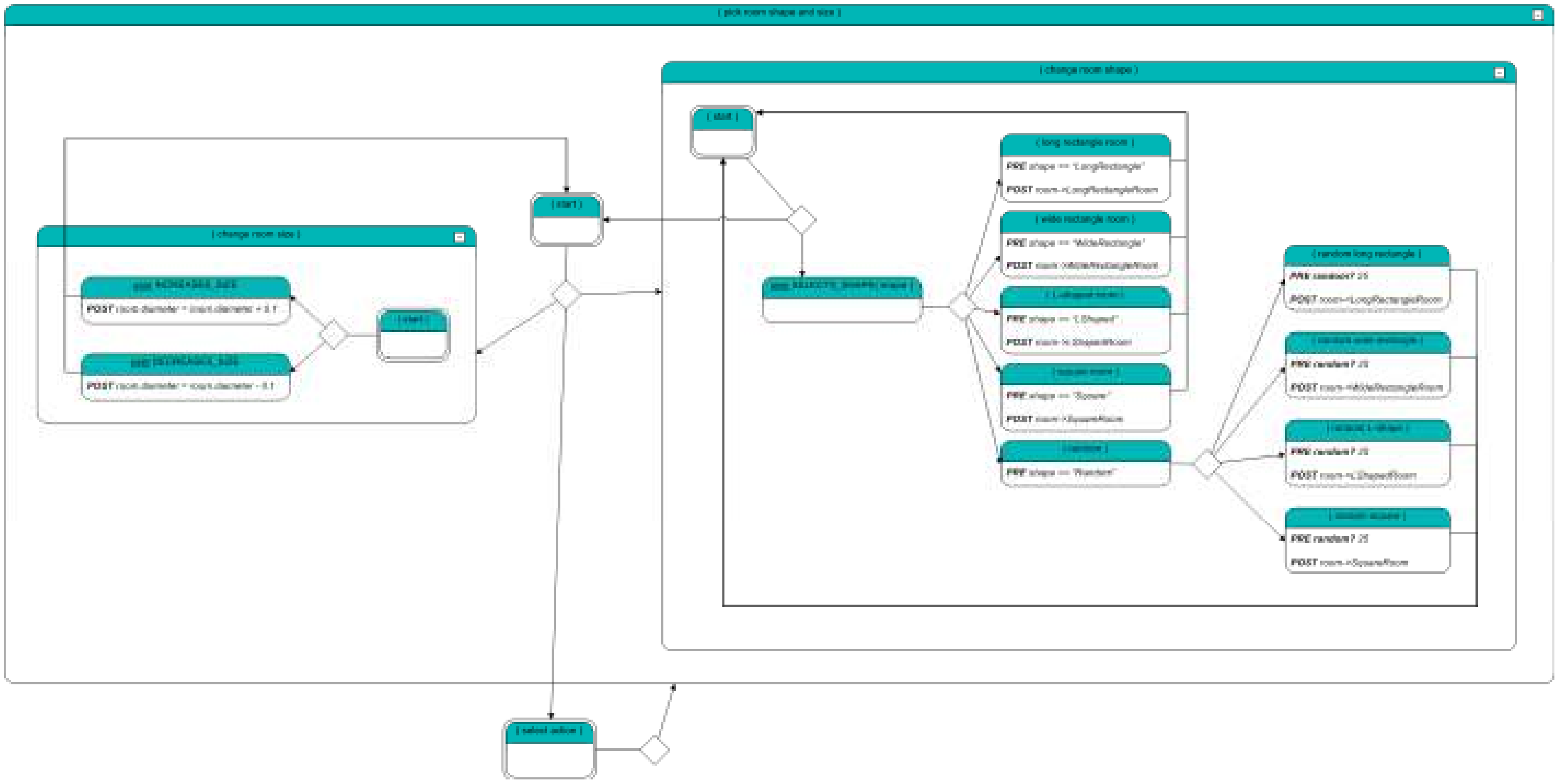
<u>**Actors**</u>
ACTOR room **role** DecoratingRoom **playedBy** room1
ACTOR n **role** DecorativeFurniture

ACTORS furniture **role** DecorativeFurniture
ACTORS selectedFurniture **role** DecorativeFurniture

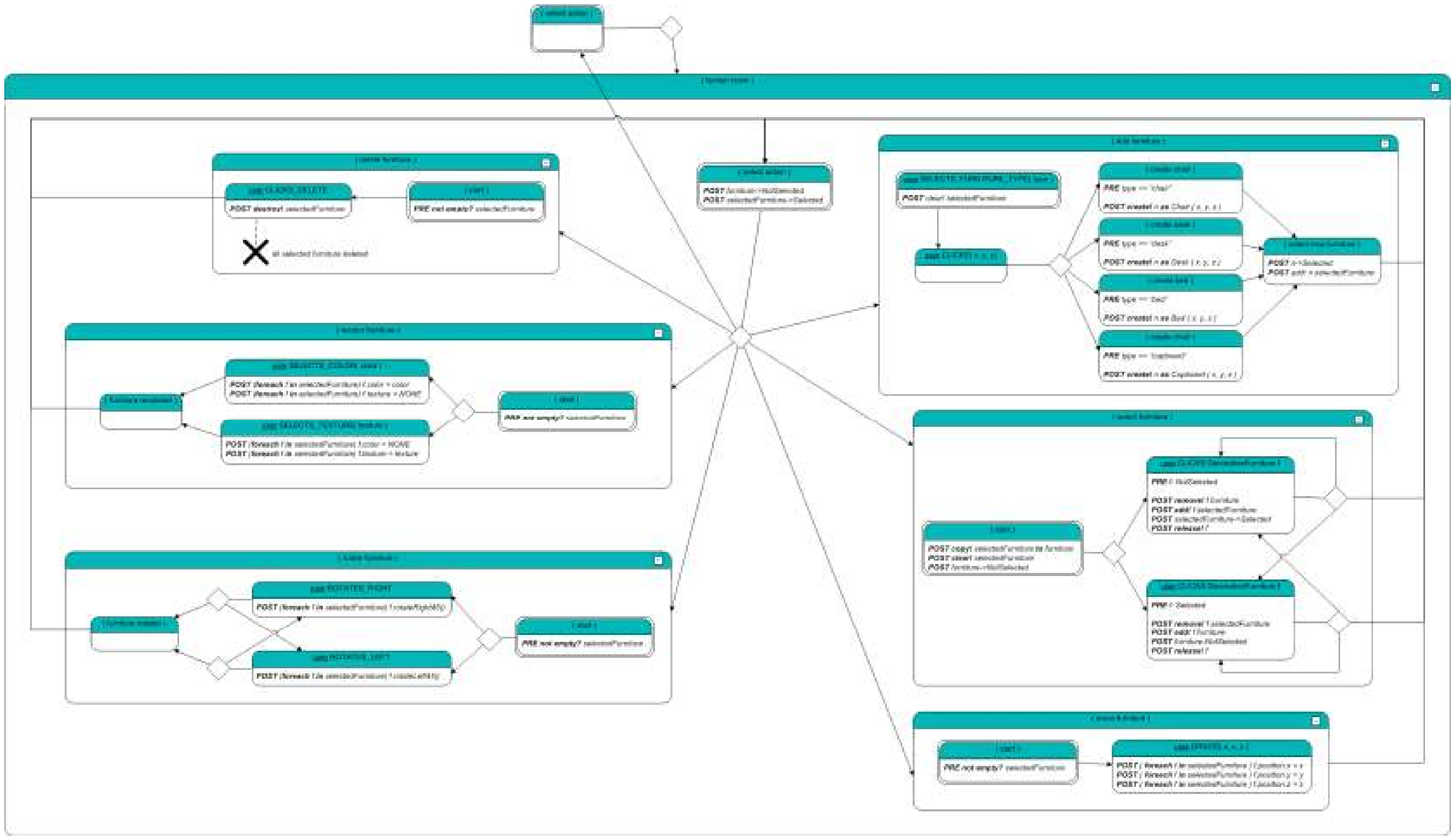
B.2. Choose Wall Color or Wallpaper


(choose wall color or wallpaper)
(start)
user SELECTS_COLOR(color)
POST room.wallColor = color
user SELECTS_WALLPAPER(wallpaper)
POST room.wallTexture = wallpaper
(select action)

B.3. Pick Room Shape and Size



B.4. Furnish Room

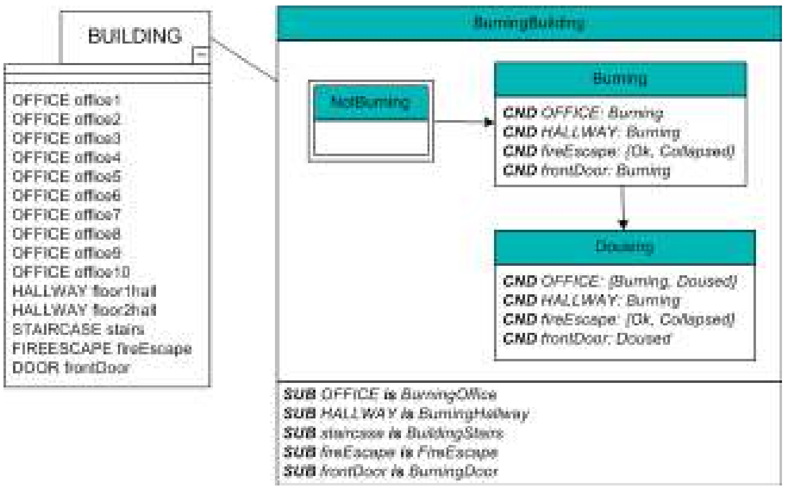


POST destroy! selectedFurniture
PRE not empty? selectedFurniture
all selected furniture deleted
POST furniture->NotSelected
POST selectedFurniture->Selected
POST clear! selectedFurniture
PRE type == "chair"
POST created n as Chair (x, y, z)
PRE type == "desk"
POST created n as Desk (x, y, z)
PRE type == "bed"
POST created n as Bed (x, y, z)
PRE type == "cupboard"
POST created n as Cupboard (x, y, z)
POST n->Selected
POST add! n selectedFurniture
PRE f NotSelected
POST remove! f furniture
POST add! f selectedFurniture
POST selectedFurniture->Selected
POST release! f
PRE f Selected
POST remove! f selectedFurniture
POST add! f furniture
POST furniture->NotSelected
POST copy! selectedFurniture to furniture
POST clear! selectedFurniture
POST furniture->NotSelected

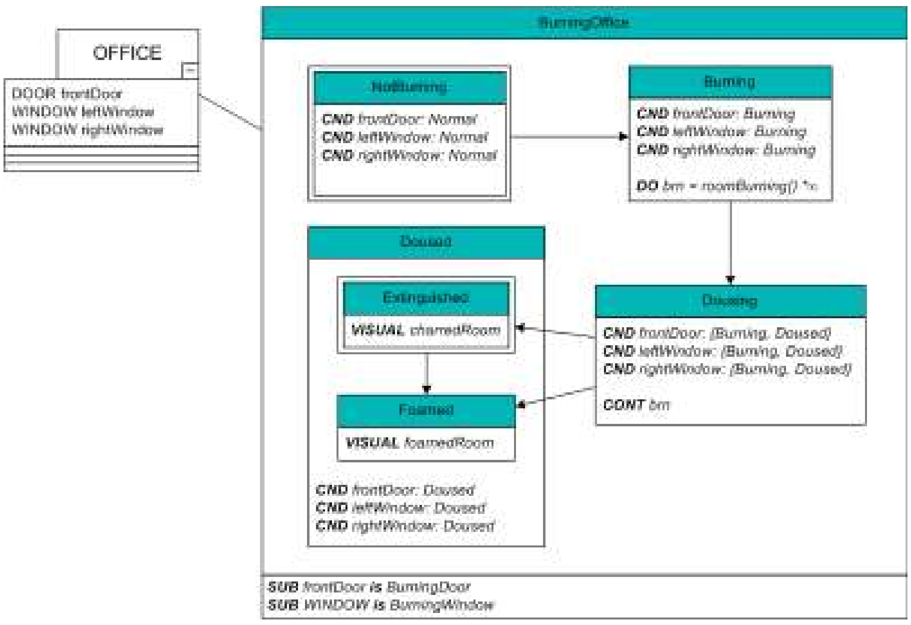
Appendix C

Fire Alert Example

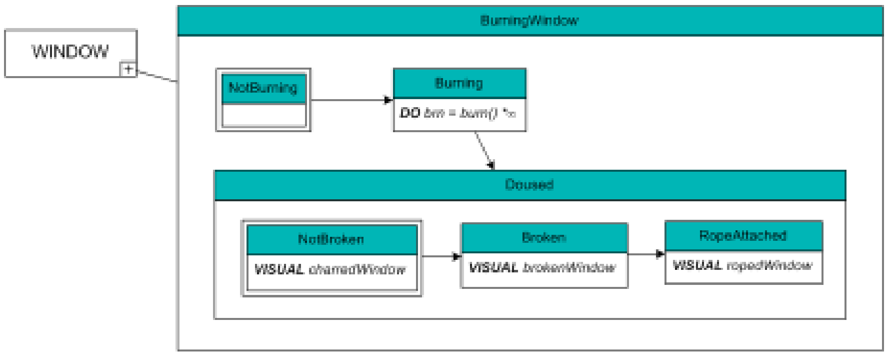
C.1. Building Actor Graph



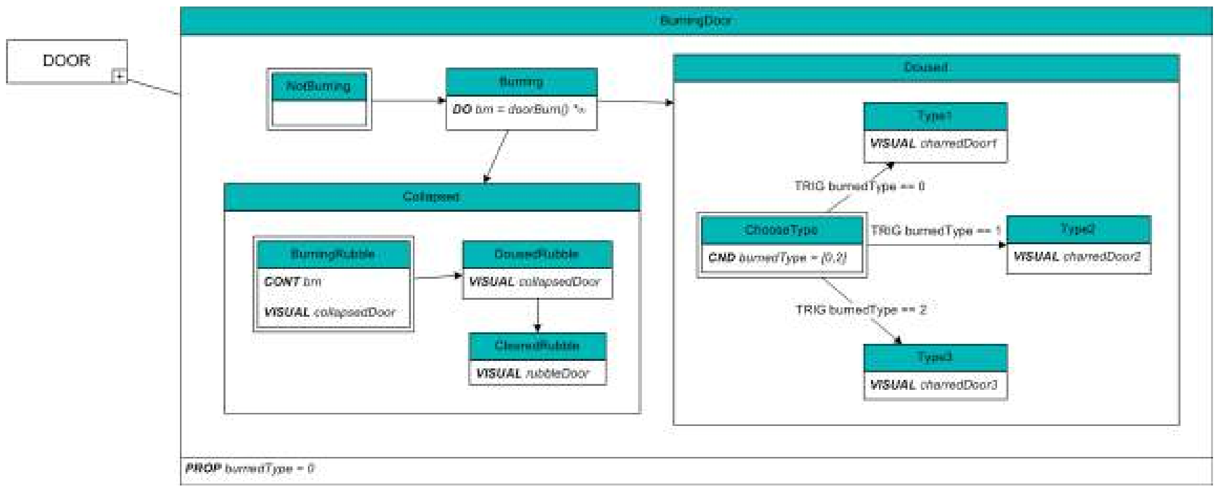
C.2. Office Actor Graph



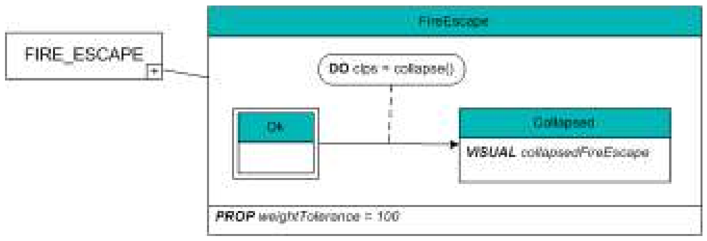
C.3. Window Actor Graph



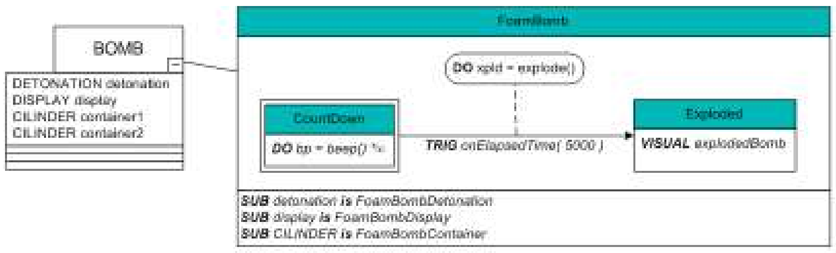
C.4. Door Actor Graph

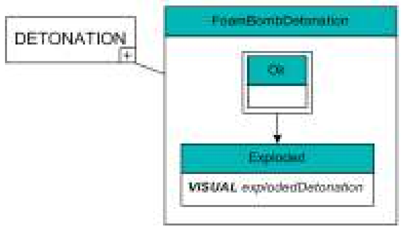


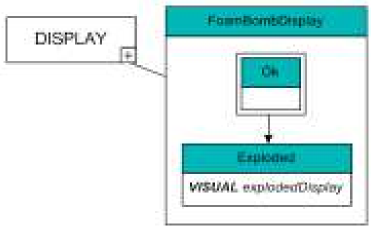
C.5. Fire-Escape Actor Graph

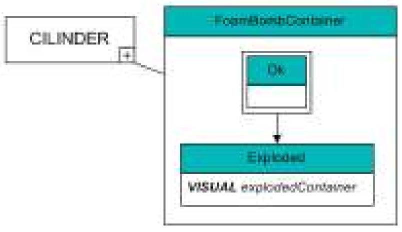


C.6. Foam Bomb Actor Graphs

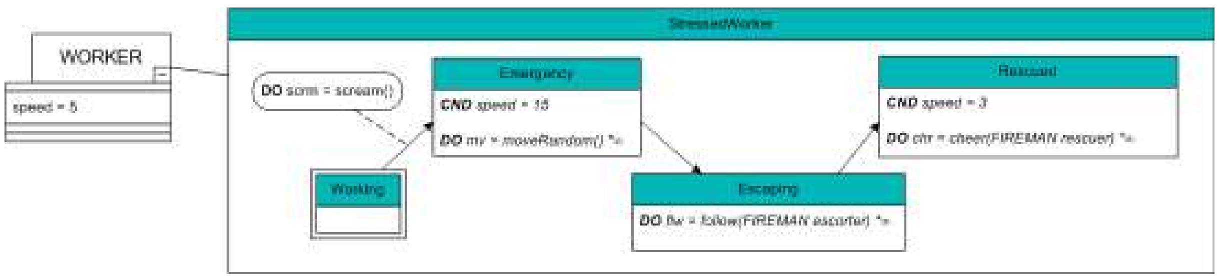




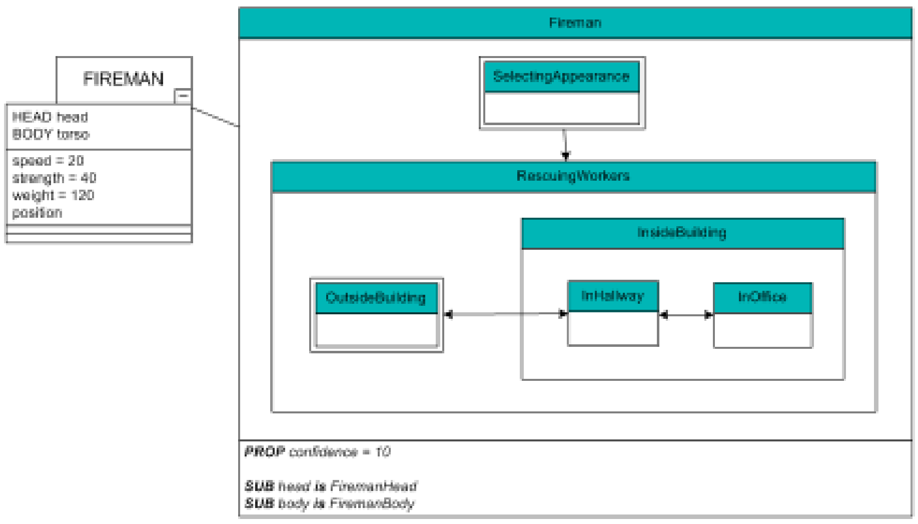


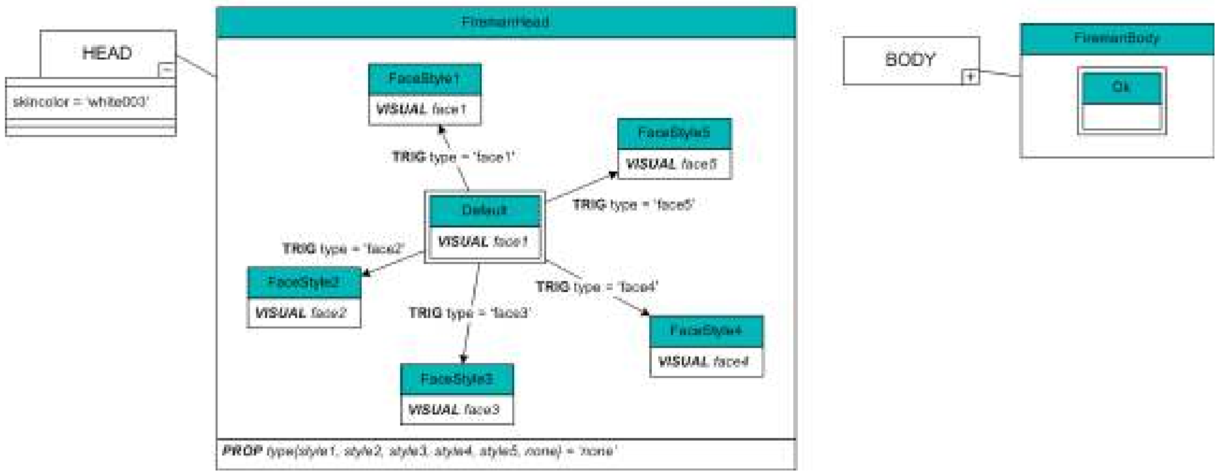


C.7. Worker Actor Graph



C.8. Fireman Actor Graphs



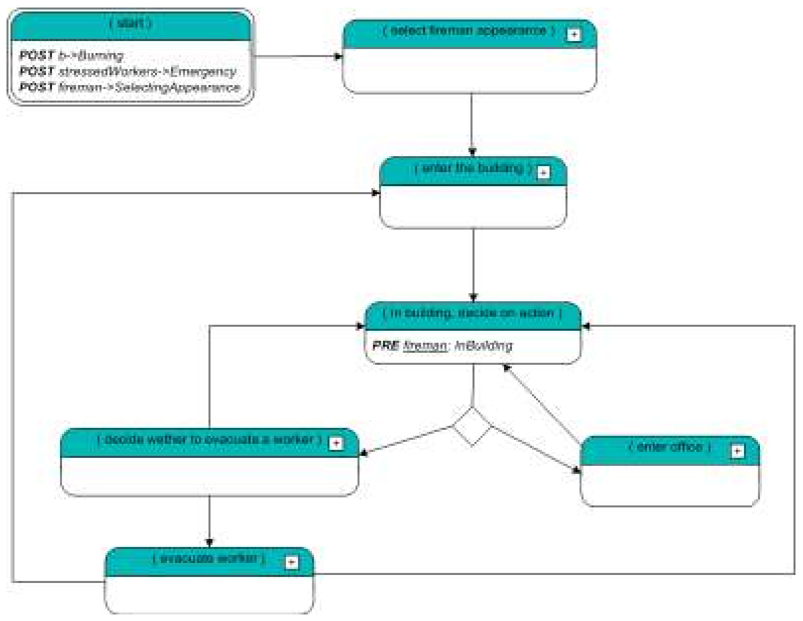


C.9. Actor Definitions

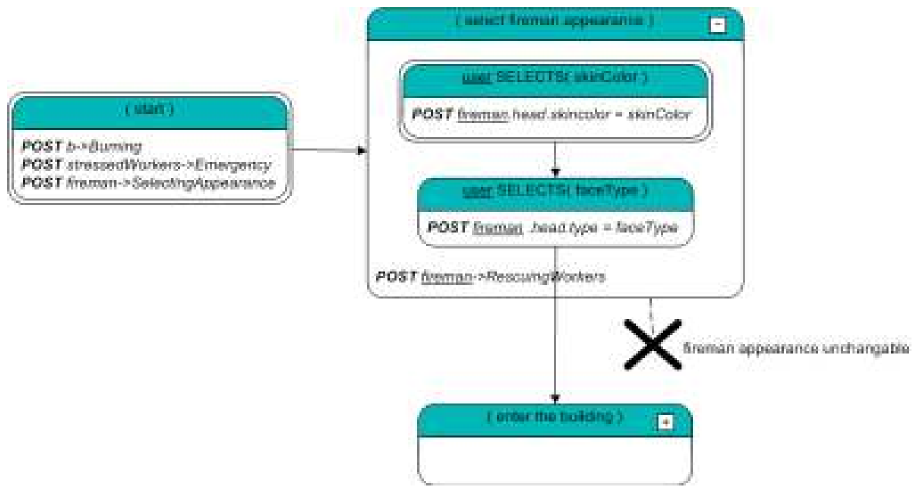
```
ACTORS
ACTOR b role BurningBuilding playedBy b1
ACTOR fireman role Fireman playedBy fireman1
ACTOR foamBomb role FoamBomb

ACTORS stressedWorkers role StressedWorker[10] playedBy
        [sw1, sw2, sw3, sw4, sw5, sw6, sw7, sw8, sw9, sw10]
ACTORS rescuedWorkers role StressedWorker[]
```

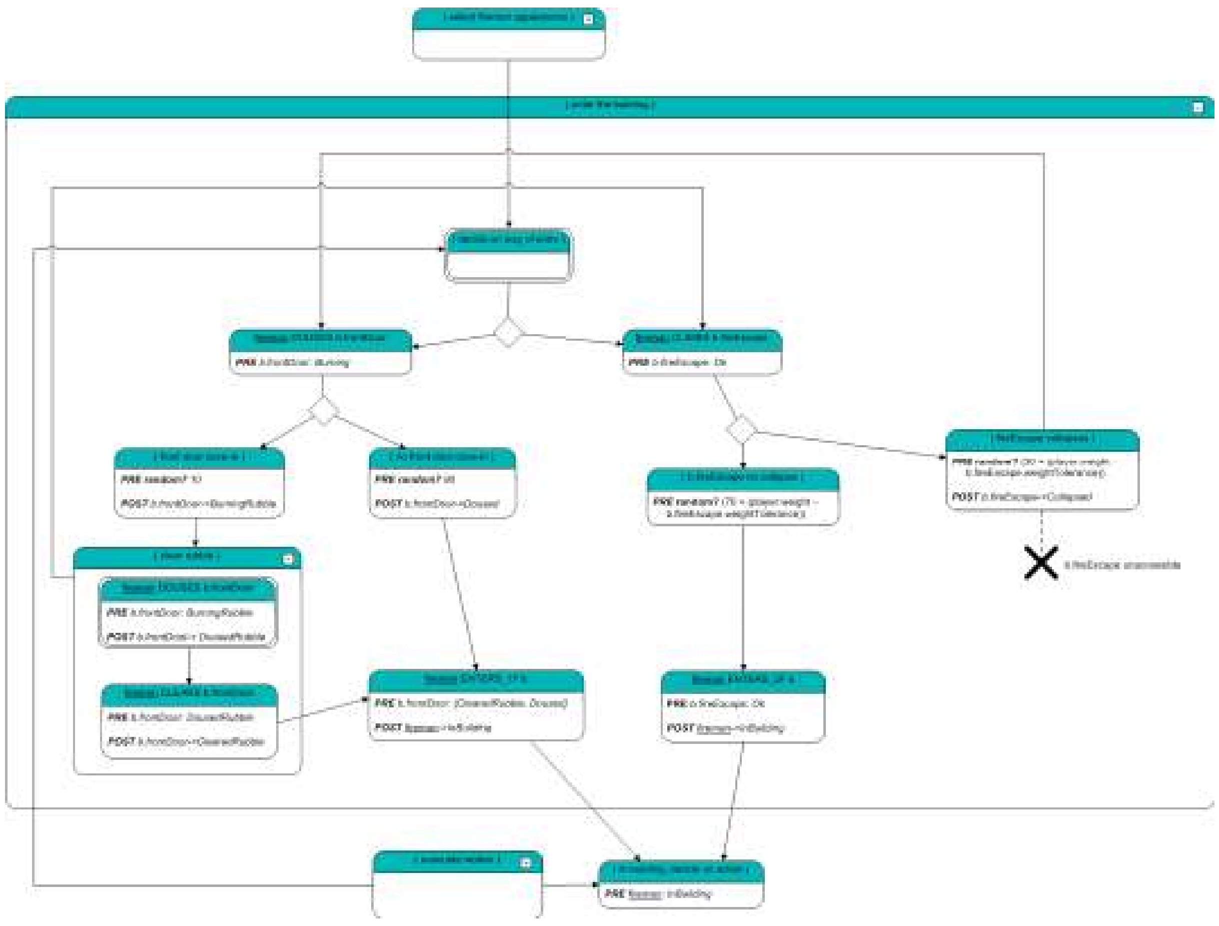
C.10. Fire Scenario Graph : Top Level


(start)
POST b->Burning
POST stressedWorkers->Emergency
POST fireman->SelectingAppearance
(select fireman appearance)
(enter the building)
(in building, decide on action)
PRE fireman: InBuilding
(decide wether to evacuate a worker)
(enter office)
(evacuate worker)

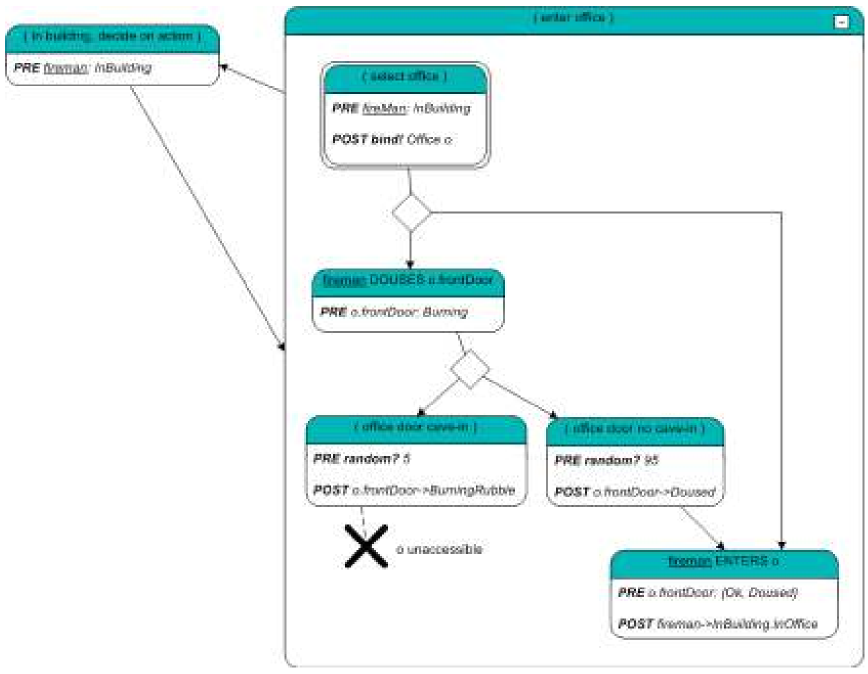
C.11. Fire Scenario Graph : Select Fireman Appearance


(start)
POST b->Burning
POST stressedWorkers->Emergency
POST fireman->SelectingAppearance
(select fireman appearance)
user SELECTS(skinColor)
POST fireman.head.skincolor = skinColor
user SELECTS(faceType)
POST fireman .head.type = faceType
POST fireman->RescuingWorkers
fireman appearance unchangable
(enter the building)

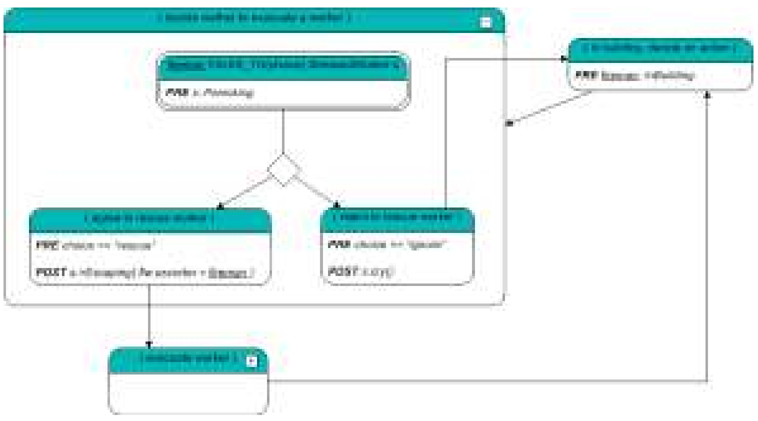
C.12. Fire Scenario Graph : Enter Building


PRE random? 10
POST b.frontDoor->BurningRubble
PRE b.frontDoor: BurningRubble
POST b.frontDoor-> DousedRubble
PRE b.frontDoor: DousedRubble
POST b.frontDoor->ClearedRubble
PRE b.frontDoor: (ClearedRubble, Doused)
POST fireman->InBuilding
PRE b.fireEscape: Ok
POST fireman->InBuilding
POST b.fireEscape->Collapsed
b.fireEscape unaccessible
PRE fireman: InBuilding

C.13. Fire Scenario Graph : Enter Office



C.14. Fire Scenario Graph : Evacuation Decision



C.15. Scenario Graph : Evacuate Worker