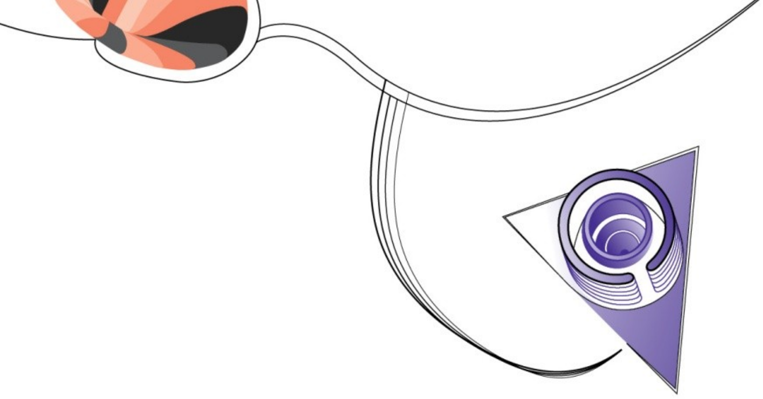
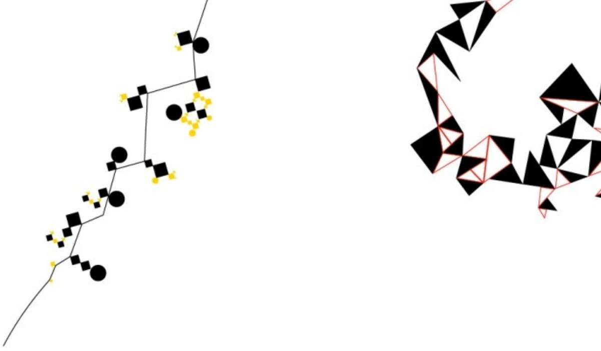
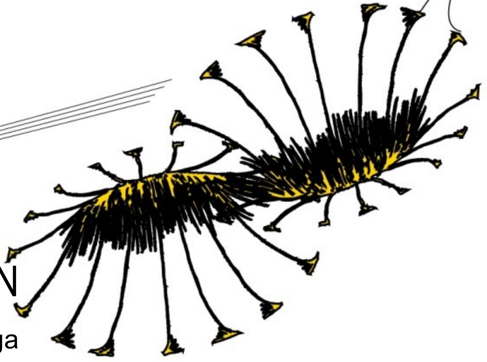




# IMPROVING COMMUNICATION BETWEEN COMPUTER PROGRAMMERS AND DOMAIN EXPERTS.

## CREATING A GENERIC SCENARIO DESCRIPTION LANGUAGE TO AID THE PROCESS OF DESCRIBING SCENARIOS FOR EMERGENCY RESPONSE TRAINING SIMULATORS.



MASTER DISSERTATION

BSc. Jesper Jeeninga

GRADUATION COMMITTEE

Dr. Job Zwiers

Dr. ir. Dick Quartel

MSc. Mark ter Maat

Dr. ir. Marten van Sinderen

UNIVERSITEIT TWENTE.

novay

NETWORKED INNOVATION

# Abstract

The communication process between programmers and domain experts is time consuming. This is caused by a “knowledge gap” between the two parties. In this case we examine the emergency response training simulation domain. Domain experts deliver domain specific information to the programmer. The programmers use this information to build an emergency response training simulator.

There is no description language available to describe such a scenario. We introduce GSDL (Generic Scenario Description Language), creating a general solution for scenario description. We claim this will reduce the communication problems and increase the efficacy of scenario generation.

To come close to a complete solution we suggest a graphical Generic Scenario Description Tool based on GSDL.

## Acknowledgments

I would like to show my gratitude to the people who helped me complete my master program. First of all to my supervisors Dick Quartel and Job Zwiers who I know to have put more time in me then they could spare. I appreciate they let me figure things out myself instead of just telling me what to do. This was hard for me but I learned a lot from it.

Then to Mark ter Maat and Marten van Sinderen who agreed to join the committee on the last moment and offered some of their private time to read my report and help me finish on time.

I would also like to thank Jan Schut, who helped me during my whole study. One appointment with him always helped me straighten things out and have the motivation to carry on.

I want to thank my parents Ate and Annie Jeeninga who I know to support me in everything I do. They are the best parents anyone could wish for. I would also like to thank my sister Rinske and her boyfriend Siebe Bosselaar for their support and time to provide feedback on my report.

Special thanks to my colleagues at Novay and the HMI research lab. We had a great time together and I made some special new friends.

This dissertation would not have been possible unless the following people shared their knowledge with me: M. Hendrix & F. Brederode (E-semble), E. Kramer, M. Mensink & A. Osinga (Trimm), D. Nusman (Re-Lion), P. Porskamp & T. de Groot (T-Xchange), E. Widding (Fire Department Enschede), B. Kobossen (NIFV), N. Wijngaards (DECIS), T. Benjamins (TU-Delft).

Also special thanks go to Alex Champandard and his wife Petra whose hospitality I enjoyed during the AI-GameDev Conference in Paris.

## Contents

# 1 Introduction

## 1.1 Problem Description

Earlier research shows us there are many emergency simulators available (Jeeninga, 2010). All of which have different implementations. These simulators try to achieve the same goal: offer a cost-efficient and safe training environment for emergency personnel; allowing police, medics and firefighters to train dangerous missions in a safe virtual environment. This has proven to be a successful addition to their training program.

We discovered a shared problem during the research about emergency simulators (Jeeninga, 2010). During the interviews, each respondent admitted the fact that the users lack the knowledge for efficient use of the simulator and the programmers lack the knowledge to program a simulator efficiently. (Hendrix & Brederode, 2009)(Kobossen, 2010)(Kramer, Mensink, & Osinga, 2009)(Porskamp & Groot, 2009)(Widding, 2009) (Nusman, 2009). This is a specific problem but it exists in other domains as well (For example, the medical software development area (Shortliffe, 2006)). The "How projects really work" cartoon[1] shows the general problem.

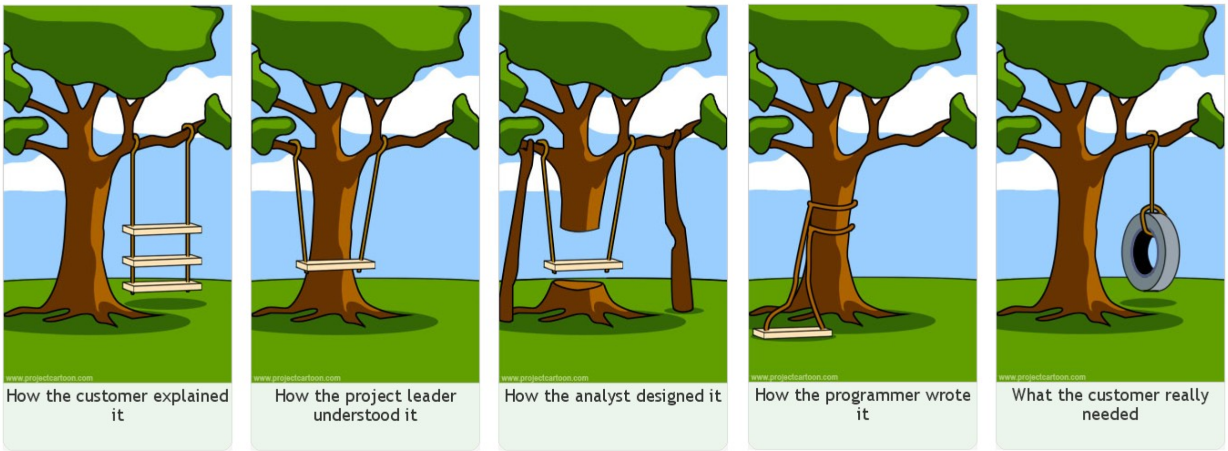

www.projectcartoon.com
How the customer explained it
www.projectcartoon.com
How the project leader understood it
www.projectcartoon.com
How the analyst designed it
www.projectcartoon.com
How the programmer wrote it
www.projectcartoon.com
What the customer really needed

**Figure 1 - The "How projects really work" cartoon.**

With emergency simulators the main problem is not the use, but the generation of a single scenario. The people who want to create a training scenario (we call them domain experts) have the knowledge about the scenario, but they lack the knowledge to import this in a simulator. A simulation programmer holds the knowledge to implement this scenario but lacks the expertise about the emergency response domain. We spotted a gap in knowledge needed by both parties. In addition the domain expert does not exactly know what part of his information the programmer needs to implement the scenario. Figure 2 depicts a visual representation of this problem.

[1] http://www.projectcartoon.com/ (visited 2010)

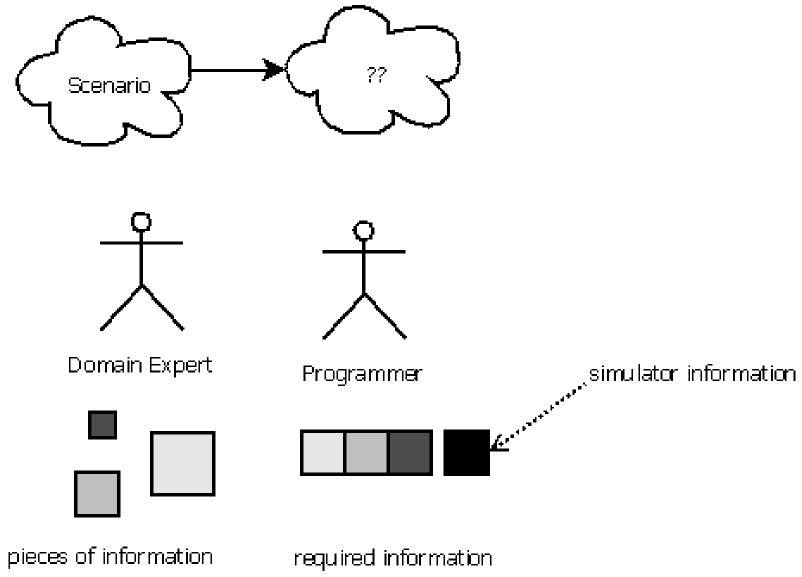


**Figure 2 – Picture of the knowledge gap problem.**

The most common solution to this problem is to have intensive contact between both parties. They have to meet often and discuss each part of the scenarios individually. The programmer applies an application life cycle; each software engineering model includes multiple meetings with the customer. After the first meetings the programmer starts programming, then both parties join again to review the scenario followed by another programming session and so on (prototyping). Many companies faced this problem and this solution just came naturally. In one case this solution advanced, over a time period of 8 years, where the programmers just learned enough about the domain to lessen the number of meetings. Both parties learned from each other and the problem started to dissolve. Due the physical distance of both parties it was not possible to meet in person as often as they would like. The cost of one specific scenario is still 80.000 Euros. In the long run this solution is still not efficient because new personnel have to cope with the same problem again when the current workers stop working on this project. (Kobossen, 2010)

Another company tried to overcome this problem by training the domain experts to be able to program some scenario-logic themselves. The company provided a tool to allow easy editing of the simulator code. This solution did not work out. The code was still too difficult for the domain experts to understand. Also the limits of the tool did not allow the domain experts to implement directly every scenario they would want. They would need some programming skills to work around these limits. Their programmers now use the tool themselves but they still use the first approach: intensive contact (Kramer, Mensink, & Osinga, 2009).

It is possible to train people to understand both areas. This works, for example, in the healthcare domain. The need for medical experts, who are also able to create software applications, becomes so high; new training programs arise to train people in Medical Software Engineering. The domain we are looking into is not big enough for this to happen.

To aid developing crisis simulations it is helpful to solve this problem in an efficient way. Our proposed solution is to be able to describe a scenario in a language both sides can easily understand. A language, that not only provides all the needed static information like locations of objects, but also some logic about decisions for the possible courses of action.

This knowledge gap problem does not only occur in the emergency response domain. Current times show a trend of IT-solutions for many other domains as well. Here the same problem occurs when domain experts try to explain their ideas to programmers. Our solution could be an inspiration for people with the same problems.

## 1.2 Problem Analysis

To specify the knowledge gap problem, we look at the problem from the side of the domain expert and from the other side, the programmer. We define what specific problems occur on each side.

We define the knowledge gap problem in our domain as followed:

**P0- It takes too much time and effort for a domain expert and programmer to obtain a shared understanding about implementing a scenario.**

We present sub-problems where the people who want to create training scenarios (the domain experts) lack information from the programmer to supply the right information. We also look to the other side and define what problems occur when the programmer tries to get the right information from the domain expert. If our tool can offer a solution to any of these sub-problems we are well on our way to a solution for our main problem **P0**. By asking both domain experts and simulation programmers during the interviews in our previous research we define these sub-problems. (Jeeninga, 2010)(Hendrix & Brederode, 2009)(Kobossen, 2010)(Kramer, Mensink, & Osinga, 2009)(Porskamp & Groot, 2009)(Widding, 2009) (Nusman, 2009)

Problems where the domain expert lacks information from the programmer:

**P1- Difference in jargon**.
For example, when interpreting an “object”; the programmer could think about a software object, which could point to a person. A domain expert would not understand this object oriented approach of the programmer. The domain expert would not think of a person as an object. Speaking in different languages causes errors in communication.

**P2- Missing “obvious” information**.
Information that seems obvious to the domain expert might not seem so obvious to the programmer. For example, to the domain expert it might be obvious that a tanker truck contains 1500 liters of water. When he forgets to mention this to the programmer, who might need this number for calculations, this will slow down the scenario generation process.

**P3- Undefined restrictions.**
The simulator may have physical restrictions. For example, the number of people it can display on screen at the same time. The domain expert may need to know this in advance to avoid inventing scenarios that are not able to run in the simulator unless the programmer extends the simulator.

**P4- Practical view.**
The domain experts sometimes only knows what to do or what will happen without knowing exactly why. A heated gas-tank will explode at some time. Unknown is the underlying logic about the exact temperature and pressure. A programmer might need this information but the domain expert is not the one to supply this information to the programmer.

Problems where the programmer lacks information from the domain expert:

**P1- Difference in jargon**.
This problem works both ways. For example, the Dutch firefighters use the phrase “ruim uitschakelen” (which freely translates to “big elimination”) to point out they will shut down the powerline of a railway. This extra information is unknown to the programmers and might cause misconceptions.

**P5- Allow forbidden events to happen**.
Without enough information about the used objects in the field the programmer might allow the simulator to let events happen that should not take place. For example, the simulator could allow a firefighter to use a hose alone when you need at least two people to handle it. The difference with **P2** is that this problem is a misunderstanding issue from the programmer instead of a submission problem from the domain expert.

**P6- Abundance of information.**
The important information that defines a scenario is not much. The information needed to implement a scenario becomes larger because of the compulsory

definition of all the default values. This makes the communication time-consuming. The difficulty of this problem is to gather only these pieces of essential information from the domain expert.

## 1.3 Research Objective

The objective of this dissertation is to aid the scenario generation for crisis simulators. Our main problem is the time-consuming communication between the domain expert and simulation programmer. To aid both parties we propose the use of tools to structure the communication between these two parties.

We use the knowledge gap problem (described in section 1.1) to supply us with a set of problems (described in section 1.2). We try to find a way to tackle our main problem (**P0**) and speed up the communication. We propose a solution consisting of a set of tools based on a common language between both parties; we call this language the Generic Scenario Description language or GSDL. The solution should answer to the following requirements.

**R1- The tools should be able to describe an entire scenario and support direct feedback**. For efficient communication the choice to describe only objects and the first state of the scenario is not enough. We need solutions to describe possible courses of action and as much (simulation) logic as possible for a complete description of a scenario.
Interviews pointed out that it was often unclear to the domain experts what information is useful to a programmer. On the other hand it is often unclear what information the programmer needs; the programmer would later discover he missed some of the information. The tools should be able to point out where information might be missing or where we do not use excess information. This would aid solving **P2** by showing in advance when information is missing and **P5** by giving feedback on what information we need.

**R2- The tools should provide an unambiguous description of the scenario**. It should provide a clear answer to all the questions a domain expert could answer about the scenario. We should confine the mistakes in communication to a minimum. This would tackle **P1** and **P5** by taking away the possibilities for misinterpretation.

**R3- The tools should allow a programmer to extend or restrict the template of a scenario.** This requirement mostly goes for situations where the simulator is already present. Then the scenario programmer should be able to communicate limits or default information before the scenario generation. This will help to minimize **P3.**

**R4- The tools should allow intuitive and extensive usability for the domain expert**. The tool should not restrict the possibilities of the scenario. It should be possible to create a scenario quickly and everything a domain expert would want should be possible to add. We need to keep the domain expert from having to make difficult programming decisions. This requirement aids to reduce problems like **P1** by allowing adding extensive possibilities to scenarios. And this will partially reduce **P4**[2] by allowing the domain expert to add complex information, or a simple version if this information is unknown. It should not be a contradiction to **R3.** The programmer might add restrictions, but this should only present a warning or advice.

**R5- The tools should be modular and reuse previously or predefined information.** This way each scenario is just a small subset of changes instead of a load of new information. This would help to lessen time lost on **P6**. (And it would speed up the process of creating more than one scenario thus tackling **P0** in this case).

**R6- The tools should allow confirmation to both sides to confirm a similar understanding.** Some sort of back-and-forth communication method should be applied to allow both sides to confirm the simulation works as they have in mind. This combines **R2** and **R3**. The system could provide part of the feedback. Some acknowledgements need to be done in different steps during the development phase of the scenario.

## 1.4 Approach

In our previous research we created a review of current emergency response simulators (Jeeninga, 2010). We interviewed domain experts (Kobossen, 2010)(Widding, 2009) and simulation programmers (Hendrix & Brederode, 2009)(Kramer, Mensink, & Osinga, 2009)(Porskamp & Groot, 2009) (Nusman, 2009) about the problem described in section 1.1. The goal of that research was to find where we could improve emergency simulators. One of the ways to improve scenario simulation is by offering solutions for the knowledge gap problem which we defined in section 1.2.

In the end we want to find the answers to the following research questions:

**Q1-** What problems define the knowledge gap problem with scenario generation for a computer simulation?
**Q2-** What existing tools could offer a solution to any of these problems?
**Q3-** Could (a combination of) these tools offer a solution to the general knowledge gap problem? How do these tools cope with the individual problems?
**Q4-** How can we test effectiveness of the proposed tools?

Using the information from the previous research we defined the knowledge gap problem by defining groups of smaller problems that are the roots to the problem in section 1.2 (answering **Q1**). Any of these problems we can solve will contribute to the general solution. We propose a solution that uses a description language supported by tools as intermediary between the domain expert and the

[2] The problem **P4** can only be solved by a third party that knows about the internal calculations of objects. For now it would suffice if the domain expert would be allowed to just enter the practical information (**R4**) and let the programmer find out the specifics.

simulation programmer. Using the problem definitions we defined a set of requirements for our tools in section 1.3.

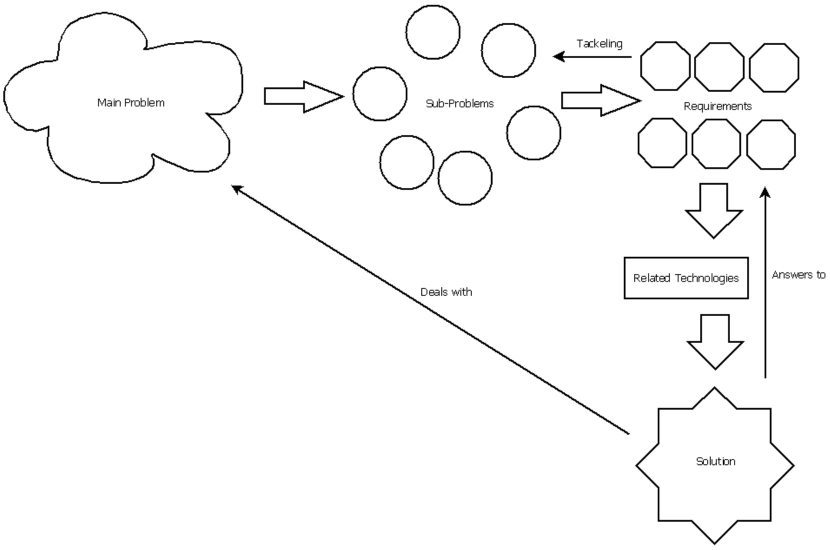


Figure 3 - Problem solving approach

We make a literature survey of possible description languages and related technologies to act as a base for our description language (to answer **Q2**). Then we propose a new solution using the best features of the existing description languages in section 3 answering **Q3**. We evaluate our findings in section 5 and draft testing methods of our solution in section 0 (answering **Q4**).

Some of the requirements cannot be met by a common language (GSDL) alone. Since we want to we base this language on complicated planning and programming languages we need extra tools to make this understandable for a domain expert (**R4**).

To support feedback (**R2**) we use a GSDL simulator. GSDL could in some cases be parsed to provide direct input for the actual simulator. The tools are depicted in Figure 4. The programmer could be able to understand GSDL code directly, but can also use the GSDL simulator and even the Scenario Description Tool.

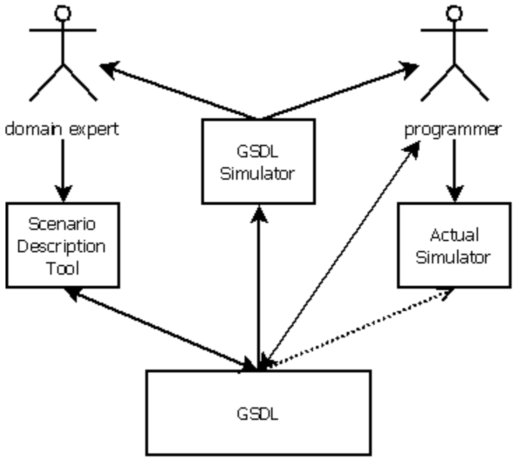


Figure 4 - GSDL Tools overview

We describe the development of GSDL in section 3.4, the GSDL simulator in section 3.5 and the Scenario Description Tool in section 4.

## 1.5 Summary

We want to answer the following research questions:

**Q1-** What problems define the knowledge gap problem with scenario generation for a computer simulation?
**Q2-** What existing tools could offer a solution to any of these problems?
**Q3-** Could (a combination of) these tools offer a solution to the general knowledge gap problem? How do these tools cope with the individual problems?
**Q4-** How can we test effectiveness of the proposed tools?

We defined the knowledge gap in section 1.2. We use the requirements provided in section 1.3 to build a set of tools. One tool is a Generic Scenario Description Language (GSDL, described in section 3) that acts as a common language between both parties. The second tool is the GSDL Simulator that provides feedback about the written GSDL code. The third tool is a user interface that provides the domain expert a usable environment to create a GSDL description of a scenario. An overview of these tools is depicted in Figure 4.

We try to solve the following communication and understanding problems:

**P0- It takes too much time and effort for a domain expert and programmer to obtain a shared understanding about implementing a scenario**
**P1- Difference in jargon**
**P2- Missing "obvious" information**
**P3- Undefined restrictions**
**P4- Practical point of view**
**P5- Allow forbidden events to happen**
**P6- Abundance of information.**

The tools should …

**R1- describe the entire scenario and support feedback;**
**R2- provide an unambiguous description of the desired scenario;**
**R3- allow a programmer to extend or restrict the template of a scenario;**
**R4- allow intuitive and extensive usability for the domain expert;**
**R5- be modular and reuse previously or predefined information;**
**R6- allow confirmation to both sides to confirm a similar understanding.**

The table below specifies what requirement would help reducing which problem.

| | **P1** | **P2** | **P3** | **P4** | **P5** | **P6** |
|---|---|---|---|---|---|---|
| **R1** | | X | | | X | |
| **R2** | X | | | | X | |
| **R3** | | | X | | | |
| **R4** | X | | | X | | |
| **R5** | | | | | | X |
| **R6** | X | | X | | X | |

**Table 1 – Requirement versus Problem Table**

We assume that by living up to the following requirements, we will progress towards the main goal: speeding up the communication between the domain expert and the simulation programmer (**P0**).

We created two of the three major tools of our solution. The GSDL language (section 3.4) and the GSDL simulator (section 3.5).

## 1.6 Overview

We discuss the background information in section 2. First we look at ideas from distant domains like scenario description for movies and games. Following, we look more into existent solutions for description and decision making tools.

We define the following 3 tools:

- Generic Scenario Description Language (GSDL), a script language that can describe scenarios in an efficient way. (section 3.4)
- GSDL Simulator, a simple simulator that uses GSDL to support testing of the scenarios described in GSDL. (section 3.5)
- Scenario Description Tool, a tool to support creation of GSDL scenarios. (section 4).

For an overview of the tools see Figure 4 on page 12.

We start describing a simple scenario, using the ideas of our studies, in section 3. This provides us a description method we use as a start for GSDL.

This narrows the knowledge gap problem because of the following advantages of using these tools:

- There is less need for appointments with the domain expert and programmer
- There is less time needed to program the simulator
- The quality of the information delivered by the domain expert is better.

We evaluate the results and issues in section 5 and we elaborate our conclusions in section 6.

# 2 Background information

We will have a look at similar problems in different domains in section 2.1. Then we will look into existing scenario description languages in section 2.2. During this research we found that planning languages are capable of describing a scenario to find a solution for a specific problem. We decided to take a better look at them in section 2.3. We discuss the conclusions about the background information in section 2.4.

## 2.1 Related domains

### 2.1.1 Introduction

We define related domains as domains that do not directly use scenarios in a way we plan to do, but might be worth looking at, like movie or game scenarios. Section 2.1.4 describes what lessons we learned from the related domains.

### 2.1.2 Scenario description in movies

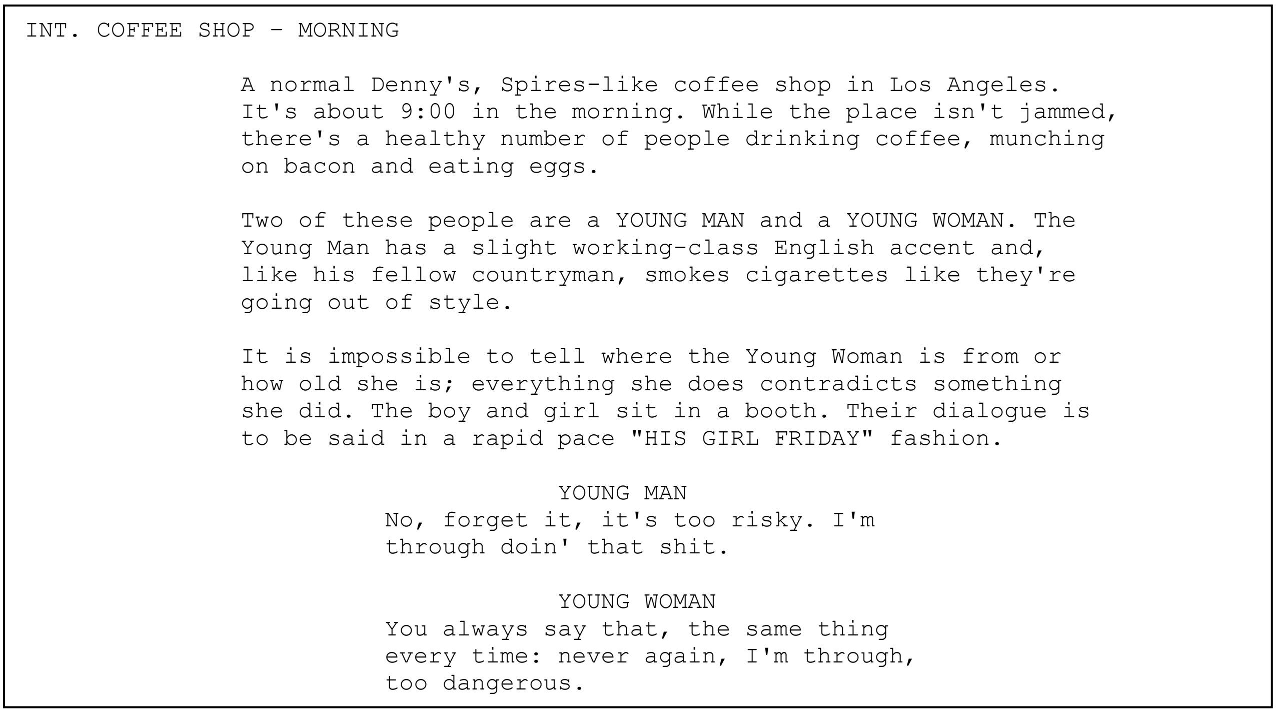
Moviemaking is a related area where they use scenarios. The more often used term in this field is a *screenplay*. It is relevant to our research to check to what extend we can use techniques from screenplays for our method of creating scenarios.

A *script* describes a screenplay. In this area writing a script is a separate profession called *screenwriting*. There are some loose rules about the style of a script. The goal is that the director can read it and understand the general idea. The directors can fill in the missing or undescribed parts by own insight. Compared to our problem we could say that we can compare a screenwriter to the domain expert and the director to the simulation programmer. The script would then be the solution we are looking for. The big difference is the number of details and fixed facts that a script describes compared with what we need to solve our problem. The programmer should not be able to use his own insights for the implementation; we want to avoid ambiguous interpretations.

A few fixed rules to managing document style bind a scenario description for a movie but the rest is open to a different interpretation. The first obvious rule is the use of a so called *slugline* before a scene. This is a capitalized heading and it provides a short description of the time and location of the scene, it usually exists of 3 parts:

1. INT./EXT. Shows whether the scenario is in- or outside
2. Location description. A brief description of the general location. For example, KITCHEN
3. Time indication: for example, NIGHT.

A space, a hyphen and another space separate the location and the time. Sometimes they also add camera hints to this line. There are no strict rules about the contents or layout of the slugline. They just consider it good manners to keep to the protocol.



```
INT. COFFEE SHOP - MORNING

               A normal Denny's, Spires-like coffee shop in Los Angeles.
               It's about 9:00 in the morning. While the place isn't jammed,
               there's a healthy number of people drinking coffee, munching
               on bacon and eating eggs.

               Two of these people are a YOUNG MAN and a YOUNG WOMAN. The
               Young Man has a slight working-class English accent and,
               like his fellow countryman, smokes cigarettes like they're
               going out of style.

               It is impossible to tell where the Young Woman is from or
               how old she is; everything she does contradicts something
               she did. The boy and girl sit in a booth. Their dialogue is
               to be said in a rapid pace "HIS GIRL FRIDAY" fashion.

                                     YOUNG MAN
                         No, forget it, it's too risky. I'm
                         through doin' that shit.

                                     YOUNG WOMAN
                         You always say that, the same thing
                         every time: never again, I'm through,
                         too dangerous.
```

Figure 5 - A movie script example

Below the slugline is an objective description of the scene followed by a chronologic story of events. The screenwriter also writes the speech dialogs in a specific protocol.  They use smaller, centered lines and capitalize names.

As you can see the script shown in the example above is open for interpretation on many parts. For example, a director is supposed to know how a "Spires-like coffee shop" should look like.

### 2.1.3 Scenario description editors for games

In games, the game presents the player with different kinds of scenarios. Level editors invent these scenarios. Here is a thin line and much communication needed between the programmers and level editors. We will take a look at a selection of how editors allow level designers to create scenarios for the players. This can provide us insights on how they describe scenarios.

#### *Valve Hammer Editor*

To create levels for games like Half-Life® and any other games that use the source™ engine one can use the Valve Hammer Editor™.[3] The first purpose is to create 3D environments. Next to that it allows an editor to add extra scenario information. The Valve Hammer Editor™ can load a configuration file (Forge Game Data file or FGD file) that defines the usable objects for the editor.

[3] http://developer.valvesoftware.com (visited 2010)

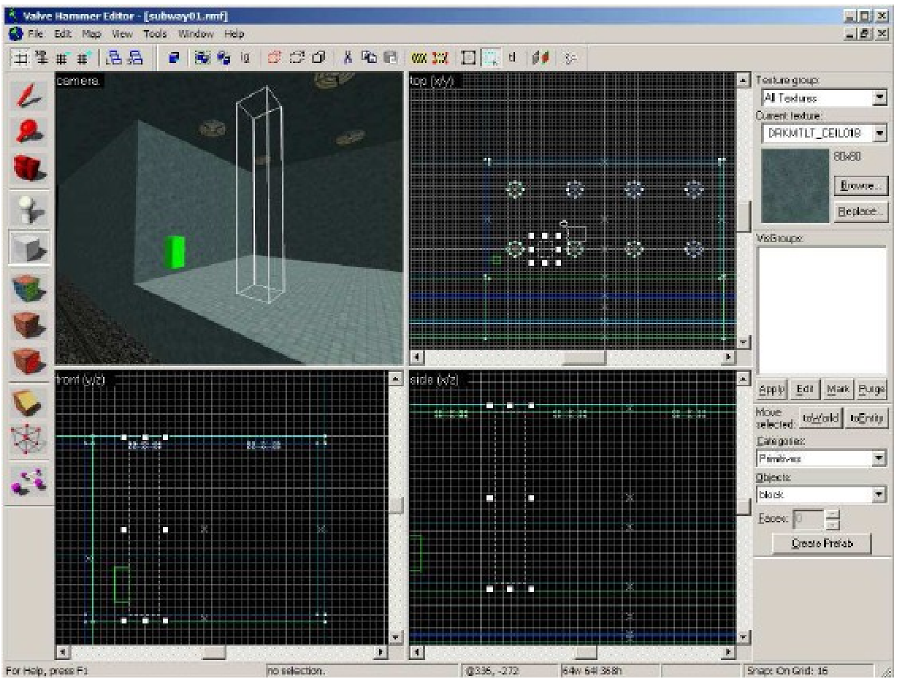


Figure 6 - Valve Hammer Editor

This FGD file is the connection between the code of the game and the properties of the level (scenario). This makes the editor usable for multiple games. Each game provides its own FGD that describes all the game-hooks. The editor can use these hooks to add interaction with the game to a level.

```
@PointClass base(Targetname, Origin) = example_entity : "example"
[
         spawnflags(flags) =
         [
                  32 : "A flag" : 0       // 0 means the flag is not-ticked by default
                  64 : "Another flag" : 1 // 1 means the flag is ticked by default
         ]

         foobarname(string) : "Name" : : "Name of foobar"
         foobargroup(string) : "Group" : "Squad1" : "Name of foobar group"
         foo(float) "Floating point number" : "100.7" : "Decimal points = fun"
         something(integer) readonly: "first number" : 0 : "This is a number which can't be
manually edited"
         something2(choices) : "second number" : 0 : "Your choice of numbers!" =
         [
                  0 : "Default"
                  1 : "Something"
                  2 : "Another Thing"
         ]

         // Outputs
         output OnSomethingHappened(void) : "Fires when something happens"
         output OnSomethingElse(void) : "Fires when something else happens"

         // Inputs
         input DoSomething(void) : "Do something"
]
```

Figure 7 - FGD example

There is a distinction between two types of objects: point based (defined by `@Pointclass`) and area based objects (defined by `@Solidclass`). A point based entity is a light source for example. When you place this object at a location in the editor; the game will know where to put the light source. Area based entities can define an area where specific rules of the game apply. For example, a trigger: the designer creates an area and if the player walks into this area a trigger will activate an alarm.

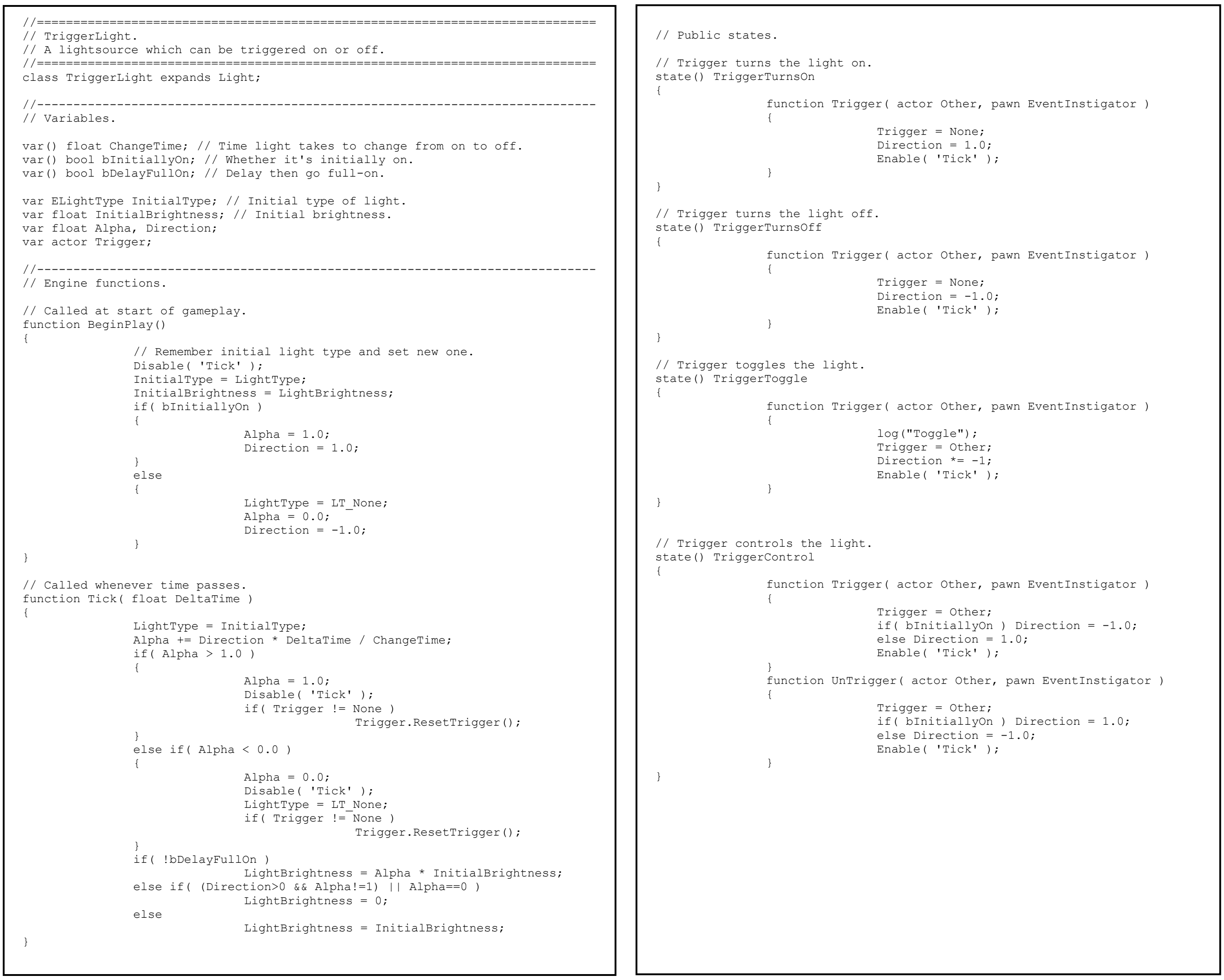
The way they use this editor in the game development industry suggests that in this area there was need for a solution to solve the knowledge gap between the programmer and the level editor. They solved this by using the FGD to hook the editor to a game. This does not specifically solve the problem, but it aids in the development. The programmer provides the hooks that limit the level editor. If there is need for new hooks the level editor should ask the programmer to add these. This step is still subject to the knowledge gap problem.

### *UnrealScript™*

For the Unreal® game engine there is a similar editor as above for the 3D environment. The main difference is there is a bigger separation between the world and the objects within. UnrealScript™ provides the objects its body using, a Java-like scripting language which allows extendible behavior. (Sweeney, 2010) The disadvantage to this is the understandability. One needs Programming experience to understand how to add behavior to objects.

The link between the engine and the scripts is the API of the game. Each scriptable object a level designer creates must extend from a base object in the game. The game calls the overrideable functions of the objects when specific conditions apply. In the example below each game-tick the game calls the function Tick. This action oriented approach is common in game scripts.

Each object can have several states. This allows the object to behave differently when in a certain state. The game calls a different function depending on the current state. This method is also common in game scripts.



```
//===========================================================================
// TriggerLight.
// A lightsource which can be triggered on or off.
//===========================================================================
class TriggerLight expands Light;

//---------------------------------------------------------------------------
// Variables.

var() float ChangeTime; // Time light takes to change from on to off.
var() bool bInitiallyOn; // Whether it's initially on.
var() bool bDelayFullOn; // Delay then go full-on.

var ELightType InitialType; // Initial type of light.
var float InitialBrightness; // Initial brightness.
var float Alpha, Direction;
var actor Trigger;

//---------------------------------------------------------------------------
// Engine functions.

// Called at start of gameplay.
function BeginPlay()
{
               // Remember initial light type and set new one.
               Disable( 'Tick' );
               InitialType = LightType;
               InitialBrightness = LightBrightness;
               if( bInitiallyOn )
               {
                              Alpha = 1.0;
                              Direction = 1.0;
               }
               else
               {
                              LightType = LT_None;
                              Alpha = 0.0;
                              Direction = -1.0;
               }
}

// Called whenever time passes.
function Tick( float DeltaTime )
{
               LightType = InitialType;
               Alpha += Direction * DeltaTime / ChangeTime;
               if( Alpha > 1.0 )
               {
                              Alpha = 1.0;
                              Disable( 'Tick' );
                              if( Trigger != None )
                                             Trigger.ResetTrigger();
               }
               else if( Alpha < 0.0 )
               {
                              Alpha = 0.0;
                              Disable( 'Tick' );
                              LightType = LT_None;
                              if( Trigger != None )
                                             Trigger.ResetTrigger();
               }
               if( !bDelayFullOn )
                              LightBrightness = Alpha * InitialBrightness;
               else if( (Direction>0 && Alpha!=1) || Alpha==0 )
                              LightBrightness = 0;
               else
                              LightBrightness = InitialBrightness;
}
```

```
// Public states.

// Trigger turns the light on.
state() TriggerTurnsOn
{
               function Trigger( actor Other, pawn EventInstigator )
               {
                              Trigger = None;
                              Direction = 1.0;
                              Enable( 'Tick' );
               }
}

// Trigger turns the light off.
state() TriggerTurnsOff
{
               function Trigger( actor Other, pawn EventInstigator )
               {
                              Trigger = None;
                              Direction = -1.0;
                              Enable( 'Tick' );
               }
}

// Trigger toggles the light.
state() TriggerToggle
{
               function Trigger( actor Other, pawn EventInstigator )
               {
                              log("Toggle");
                              Trigger = Other;
                              Direction *= -1;
                              Enable( 'Tick' );
               }
}


// Trigger controls the light.
state() TriggerControl
{
               function Trigger( actor Other, pawn EventInstigator )
               {
                              Trigger = Other;
                              if( bInitiallyOn ) Direction = -1.0;
                              else Direction = 1.0;
                              Enable( 'Tick' );
               }
               function UnTrigger( actor Other, pawn EventInstigator )
               {
                              Trigger = Other;
                              if( bInitiallyOn ) Direction = 1.0;
                              else Direction = -1.0;
                              Enable( 'Tick' );
               }
}
```

**Figure 8 - UnrealScript example**

### 2.1.4 Conclusion related technologies

Scripts describe the scenarios in movies which have few standards and leave room for own interpretation. They always define inside or outside as well as the time of day of a scenario. The rest is defined using normal language. It is necessary to leave room for the director or actors to add information to this scenario.

There is a similar knowledge gap between the scenario writer and the movie director. Only in this domain the gap is not a real problem. One could suggest the scenario writer could even deliberately leave some parts open for interpretation by the director.

For us it is important to find out what parts of a crisis simulation scenario to define and what parts to leave open for own interpretation. For example, it is important that the length of a ladder on a fire truck matches with a real one. On the other hand it might not be as important to know the exact amount of people who walk the streets. Also it could aid the scenario description to allow adding some textual information to make certain details easier to understand. We infer that for our goal it is important the description allows no room for open interpretation.

Also a great similarity exists between our research domain and the game industry. In the game industry the level designers are better schooled to work with programmers in contrast to our domain experts. This means the knowledge gap here is smaller. We found the game industry also developed some solutions for the programmer versus editor knowledge gap. They have a solution for the communication flow from the programmer to the level editor by allowing the programmer to alter the level editor's software program to allow adding only valid entries within the level. This suits **R3** to solve **P3**.

Using a script language provides extensive possibilities for the scenario but is difficult for the user to understand. A combination of script language and fixed choices could provide a solution for our problem.

## 2.2 Scenario Description

### 2.2.1 Introduction

Little research is done on scenario description for simulators. We found a few description "languages" that are able to describe scenarios. SDL, a script used in vehicle simulation programs. It provides the simulator with some artificial intelligence rules to anticipate on situations. The description language Q is a method used for evacuation simulation. Finally we discuss MSDL, a definition language to describe a tactical version of a military situation.

### 2.2.2 Scenario description languages

#### *Simula*

Simula (Dahl & Nygaard, 1965) is a programming language designed for simulations but nowadays most people see it as the root of object oriented programming languages. They created Simula to satisfy the need for precise tools for describing and simulating complex man-machine systems. They use a language that describes systems for both people and computers (through a compiler).

The latest version (Simula 67) included objects, classes, subclasses, virtual methods, co routines, discrete event simulation, and features garbage collection. We describe an example of a simulation in simula below.

Sam, Sally, and Andy are shopping for clothes. They have to share one fitting room. Each one of them is browsing the store for about 12 minutes and then uses the fitting room only for about three minutes, each following a normal distribution. A simulation of their fitting room experience is as follows:

```
Simulation Begin
   Class FittingRoom; Begin
      Ref (Head) door;
      Boolean inUse;
      Procedure request; Begin
         If inUse Then Begin
             Wait (door);
             door.First.Out;
         End;
         inUse:= True;
      End;
      Procedure leave; Begin
         inUse:= False;
         Activate door.First;
      End;
      door:- New Head;
   End;

   Procedure report (message); Text message; Begin
      OutFix (Time, 2, 0); OutText (": " & message); OutImage;
   End;

   Process Class Person (pname); Text pname; Begin
      While True Do Begin
         Hold (Normal (12, 4, u));
         report  (pname & " is requesting the fitting room");
         fittingroom1.request;
         report (pname & " has entered the fitting room");
         Hold (Normal (3, 1, u));
         fittingroom1.leave;
         report (pname & " has left the fitting room");
      End;
   End;

   Integer u;
   Ref (FittingRoom) fittingRoom1;

   fittingRoom1:- New FittingRoom;
   Activate New Person ("Sam");
   Activate New Person ("Sally");
   Activate New Person ("Andy");
   Hold (100);
End;
```

**Snippet 1 – A Simula example**

The main block is prefixed with “Simulation” to enable simulation, using the simulation package on any block. Even nesting of simulations is possible to simulate someone doing simulations.

The fitting room object uses a queue (door) for getting access to the fitting room. When someone requests the fitting room and it is in use they must wait in this queue (Wait (door)). When someone leaves the fitting room the first one (if any) is released from the queue (Activate door.first) and therefore removed from the door queue (door.First.Out).

Person is a subclass of Process and its activity is described using hold (time for browsing the store and time spent in the fitting room) and calls methods in the fitting room object for requesting and leaving the fitting room.

The main program creates all the objects and activates all the person objects to put them into the event queue. The main program holds for 100 minutes of simulated time before the program terminates.

### *SDL*

SDL (Willemsen, 2000) stands for Scenario Definition Language. It is an event-based description language for a multiagent system. A scenario director uses SDL to pick up environmental changes and notify agents accordingly. (See Figure 9)

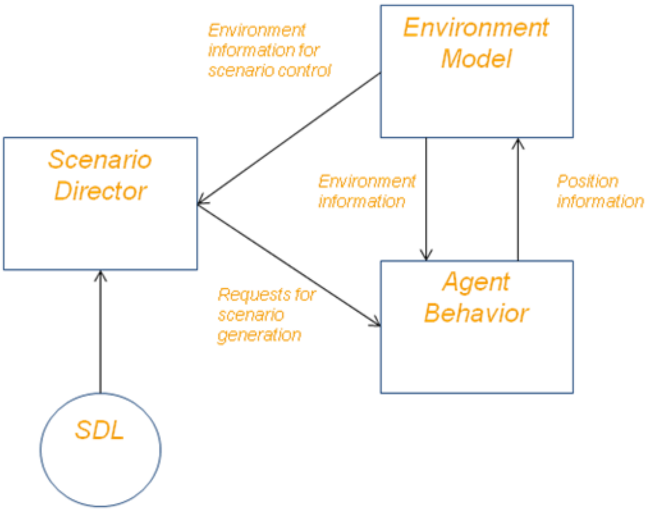


**Figure 9 - SDL outline**

The scenario director listens to the environment using monitors. When specific values are true the SDL states what behavior the agents need to fulfill. This way one cannot only describe what happens, but also when it will happen and how.

SDL is a script language using semantics from object oriented programming languages. The language is elaborate and allows complicated programming structures to define events.

```
01 defscenario LightSequencer( vehicle )
02 {
03       // Obtain a reference to the vehicle’s current road.
04       road = vehicle.queryRoad();
05       // Locate the intersection towards which the vehicle
06       // is approaching on it’s current road.
07       isect = vehicle.queryNextIntersection();
08       // Determine the traffic light controlling the next
09       // intersection towards which the vehicle is heading.
10       traffic light = isect.queryTrafficControlDevice();
11       sync point = TRANSITION TO RED;
12       aslongas (vehicle.queryDistanceToNextIntersection() > 0.0)
13       {
14               eta = vehicle.queryDistanceToNextIntersection() / vehicle.speed();
15               send traffic light sync( sync point, eta );
16       }
17       event( exit, vehicle, isect.geometry() ) triggers EXIT EVENT;
18       when( EXIT EVENT )
19       {
20               create LightSequencer( EXIT EVENT.instigator );
21       }
22 }
```

**Snippet 2 - SDL Example: every LightSequencer directs the behavior of a single traffic light to synchronize its cycle to the approach of the driver.**

The functions that are called (like vehicle.queryRoad()) need to be described in an API. These are functions of the Environment Description Framework (EDF). These dictate the boundaries of the system and thus limit the possible actions to a fixed set.

SDL is developed for driving simulators, concentrating on the fixed set of actions on driving-environments. It is possible to use the language for multiple kinds of simulators but would need a different description framework.

The SDL can send commands to objects (Snippet 2: line 15) The EDF defines these.

A monitor can listen for specific events to happen (Snippet 2: line 17). When the event occurs a trigger will be set to a true value allowing the script to respond on events.

### *Q*

The motivation to develop the Q definition language is the same as we use to develop our description language. *The Q Language … provides an interface between computing professionals and scenario writers.* (Ishida, 2005) They recognized the same knowledge gap we did (Jeeninga, 2010).

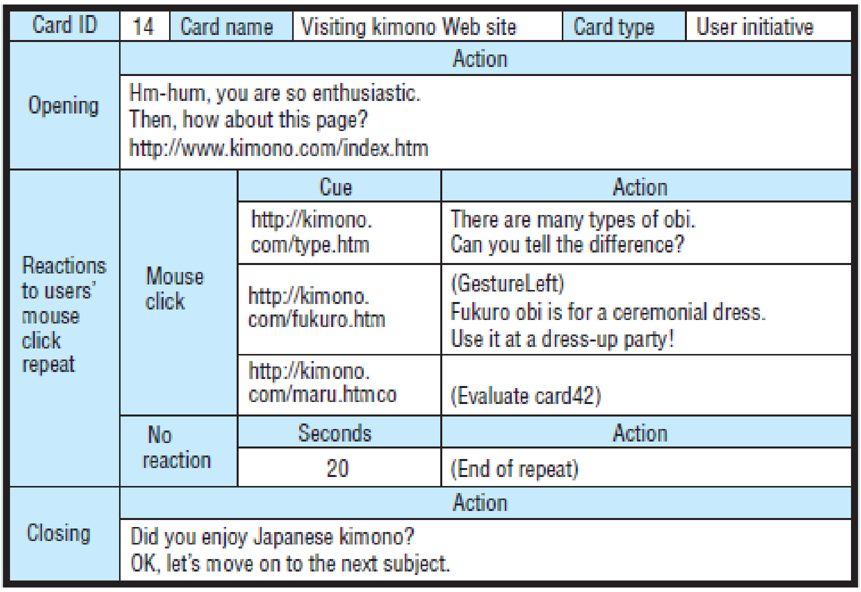
The Q language concentrates mainly on the interaction between agents. This interaction is based on cues. Each agent has one or more cues on which it will perform actions that can possibly trigger a cue by another agent. Cues are preceded by a question mark where actions are preceded by an exclamation mark. Snippet 3 shows an example of a simple scenario described with Q.

```
 (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 ...))
```

**Snippet 3 - A Q scenario example**

Snippet 3 shows the possible cues at any given time depending on the “state” of the agents using the “scene” keyword. In scene1 the possible inputs are: `?hear “Hello”` and `?hear “Bye”`. In scene2 any other input then “Hello” would advance the agents’ state to scene3. This implies a one-to-one interaction pattern without the possibility to parallelize interactions.

To make the language more accessible for less technical scenario writers they propose an Interaction Pattern Card (IPC) that describes the scenario and is convertible to the Q script language.



<table>
<tr><td>Card ID</td><td>14</td><td>Card name</td><td>Visiting kimono Web site</td><td>Card type</td><td>User initiative</td></tr>
<tr><td rowspan="2">Opening</td><td colspan="5">Action</td></tr>
<tr><td colspan="5">Hm-hum, you are so enthusiastic.<br>Then, how about this page?<br>http://www.kimono.com/index.htm</td></tr>
<tr><td rowspan="6">Reactions to users’ mouse click repeat</td><td colspan="2" rowspan="4">Mouse click</td><td>Cue</td><td colspan="2">Action</td></tr>
<tr><td>http://kimono. com/type.htm</td><td colspan="2">There are many types of obi.<br>Can you tell the difference?</td></tr>
<tr><td>http://kimono. com/fukuro.htm</td><td colspan="2">(GestureLeft)<br>Fukuro obi is for a ceremonial dress.<br>Use it at a dress-up party!</td></tr>
<tr><td>http://kimono. com/maru.htmco</td><td colspan="2">(Evaluate card42)</td></tr>
<tr><td colspan="2" rowspan="2">No reaction</td><td>Seconds</td><td colspan="2">Action</td></tr>
<tr><td>20</td><td colspan="2">(End of repeat)</td></tr>
<tr><td rowspan="2">Closing</td><td colspan="5">Action</td></tr>
<tr><td colspan="5">Did you enjoy Japanese kimono?<br>OK, let’s move on to the next subject.</td></tr>
</table>

**Figure 10 - An Interaction Pattern Card for the Q description language**

The design is as followed: the scenario writer and the programmer discuss the possible cues and actions, then the writer will write a scenario and the programmer will implement the cues and actions.

### *MSDL*

MSDL is the Military Scenario Definition Language[4], a scenario description language standardized in 2008 by the Simulation Interoperability Standards Organization (SISO). It is developed in affiliation with the US Army and designed to be able to create a common mechanism for validating and loading Military Scenarios. To allow easy access by computers or simulators they define the language using an XML schema (World Wide Web Consortium).

An MSDL scenario represents an intermediate state or a link between the planning and execution for any type of military scenario. Entities that do not support MSDL need a conversion of the plans and execution format.

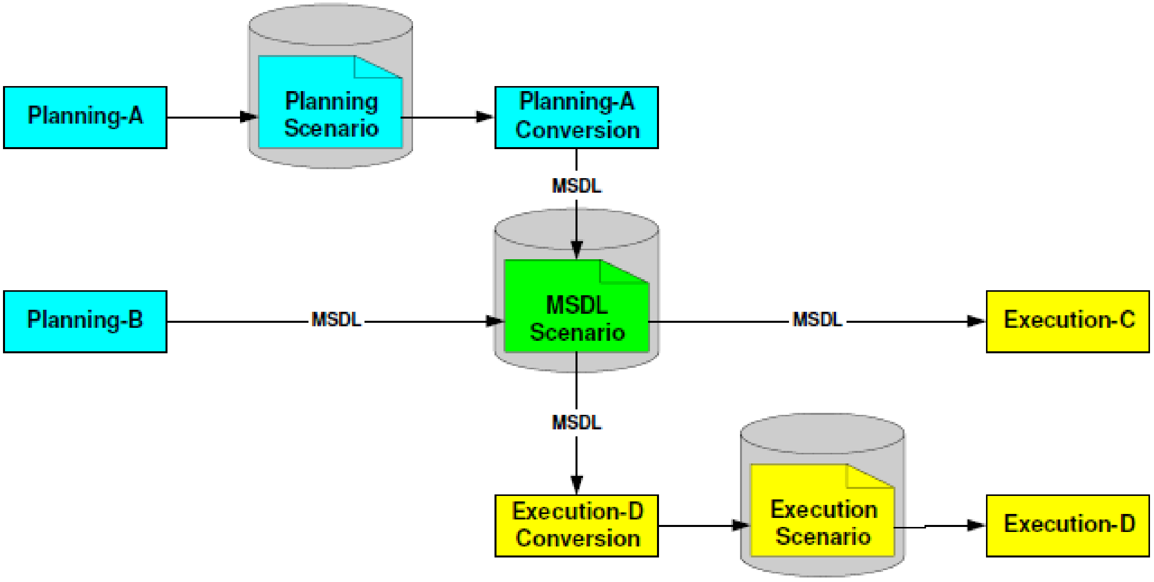


**Figure 11 - Architectural position of an MSDL Scenario**

A large default set of elements is defined that is used to describe scenarios (also see Figure 12). They call this the MSDL-Schema. The schema restricts this language to military use, but also assures that every possible military object is definable using this language. It forces defining certain variables like environmental or equipment settings and it allows adding tactical visual data like anchor points and tactical areas. This language is only used for military situations; therefore the language will be enough for these situations, as long as it is possible to define each object that might be interesting in this domain.

An MSDL Scenario exists mainly of a current situation and the possible courses of action. It is possible to create one or more plans that contain a collection of actions and events that occur simultaneously or in sequence.

[4] SISO-STD-007-2008

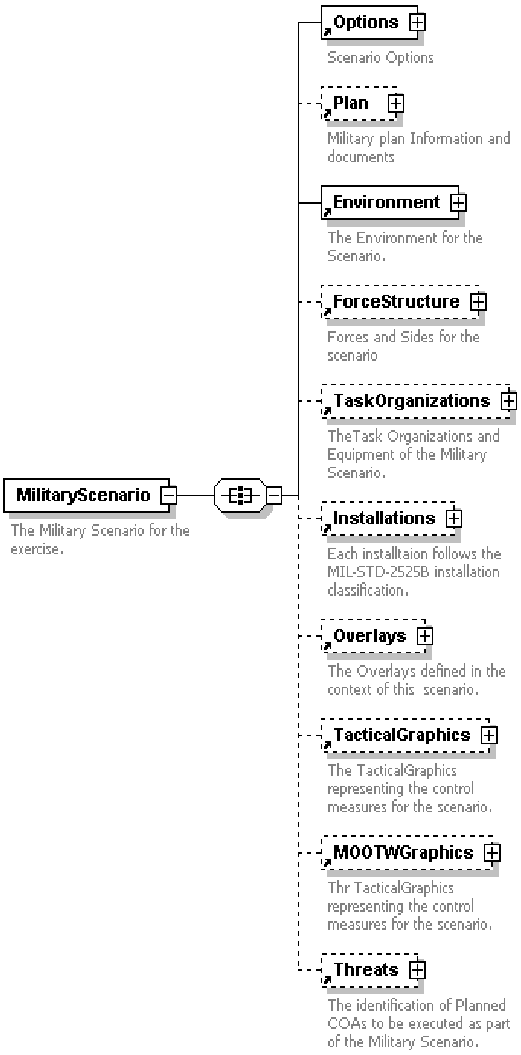

Options
Scenario Options
Plan
Military plan Information and documents
Environment
The Environment for the Scenario.
ForceStructure
Forces and Sides for the scenario
TaskOrganizations
TheTask Organizations and Equipment of the Military Scenario.
MilitaryScenario
The Military Scenario for the exercise.
Installations
Each installtaion follows the MIL-STD-2525B installation classification.
Overlays
The Overlays defined in the context of this scenario.
TacticalGraphics
The TacticalGraphics representing the control measures for the scenario.
MOOTWGraphics
Thr TacticalGraphics representing the control measures for the scenario.
Threats
The identification of Planned COAs to be executed as part of the Military Scenario.

**Figure 12 - MSDL Scenario overview**

### 2.2.3 Conclusion scenario description languages

Simula was designed for simulations and was the first object oriented programming language. This shows that object oriented programming languages are suitable for simulations. Besides being suitable for simulations, the object oriented approach also provides modularity supporting **R5**.

SDL uses programming language semantics. This allows description of behavior (supporting **R1**). It is also legible by the simulation programmer; but the semantics only resemble those of programming languages. The language still has its own keywords which the simulation programmer must understand. SDL is created to be able to optimize performance for the program; our language should be created for optimal usability for the domain expert.

SDL is too difficult for a domain expert to understand lacking support for **R4**. We considered building a user interface for SDL (like we describe in section 4 for GSDL) but the complicated programming would give us a hard task to keep scenario description simple for the domain expert. Another issue with SDL is the environment API which limits the use but this can be extended if needed. We choose for a simpler version of SDL to describe a scenario.

We can use the idea of programming semantics to add interaction and behavior into our scenarios. This would support **R1**; also an object oriented language using programming semantics also supports modularity and thus **R5.** Provision of an (extendible) API can support **R3** to define the boundaries of the simulator.

Both Q and MSDL scenario description languages describe one "state". A few choices are available which will bring you to another state. This does work in favor of the ambiguity of **R2**. We think a scenario description should not only describe loose predefined states (this does not comply with **R4**). MSDL limits the possible extensions to a predefined list; this does not allow extensive use as stated in **R4** or could restrict the domain expert too much so he cannot create a complete scenario as is required by **R1**.

| | **R1** | **R2** | **R3** | **R4** | **R5** |
|---|---|---|---|---|---|
| **Simula** | X | | | | X |
| **SDL** | X | | X | | |
| **Q** | | X | | | X |
| **MSDL** | | X | | | X |

**Table 2 – Description Language versus Requirement table**

We can use tricks from these languages to meet some requirements, but usability is a problem with description languages.

## 2.3 Planning languages

### 2.3.1 Introduction

In science we use planning languages to calculate the best solutions for planning problems. They describe an initial state of a system and a goal state. Describing such a system state is comparable with (the start of) a scenario in a simulation. Multiagent systems (Weiss, 1999) also use planning languages if there is need for some decision making. Planning languages can help make optimal decisions for agents. We can use ideas of these planning languages to describe a scenario and provide them some logic for decision making. We describe STRIPS, a planning language that acts as a base for many other types of planning languages. We describe HTN, a practicable implementation of planning languages to speed up the decision. And PDDL, an extension to STRIPS that allows the use of more complicated features.

### 2.3.2 Planning Languages

*STRIPS*

STRIPS (Stanford Research Institute Problem Solver) is an automated planner developed by Richard Fikes and Nils Nilsson in 1971 (Nilsson, 1971), later using the same name to refer to the formal language of the inputs to this planner. This language is the base for most of the languages for expressing automated planning problem instances in use today.

An STRIPS instance is comprised of:

- An initial state
- The specification of the goal states – situations which the planner is trying to reach
- A set of actions. For each action, including:
    - Preconditions (We have to satisfy this before performing the action);
    - Postconditions (what is established after performing the action).

By specifying these parameters an automated planner can calculate the (ideal) route to get from the first state to the goal state.

The conditions in strips are either true or false. Specifying nothing will assume the condition to be false.

```
Initial state: At(A), Level(low), BoxAt(C), BananasAt(B)
Goal state:    Have(Bananas)
Actions:
               // move from X to Y
               _Move(X, Y)_
               Preconditions:  At(X), Level(low)
               Postconditions: not At(X), At(Y)

               // climb up on the box
               _ClimbUp(Location)_
               Preconditions:  At(Location), BoxAt(Location), Level(low)
               Postconditions: Level(high), not Level(low)

               // climb down from the box
               _ClimbDown(Location)_
               Preconditions:  At(Location), BoxAt(Location), Level(high)
               Postconditions: Level(low), not Level(high)

               // move the box from X to Y—And the monkey moves, too
               _MoveBox(X, Y)_
               Preconditions:  At(X), BoxAt(X), Level(low)
               Postconditions: BoxAt(Y), not BoxAt(X), At(Y), not At(X)

               // take the bananas
               _TakeBananas(Location)_
               Preconditions:  At(Location), BananasAt(Location),
Level(high)
               Postconditions: Have(bananas)
```

A lot of planning languages stemmed from STRIPS. The limit to use only true or false conditions has its advantages: the problems are simple and quickly calculated. But not every problem can be described using only true or false conditions. This is why there is research to extensions of STRIPS like PDDL (described below) that would allow the generation of more complicated problems.

### *HTN*

Hierarchical Task Networks (HTNs) are used to support decision making for planning problems. It is mostly used in games with artificial intelligence. Due the limit of assigned processor calculation time, game-AI need to base decisions on optimal calculated plans. (Jeeninga, 2010).

In HTN planning, they provide the planner with a set of tasks to be performed. Then they formulate a plan by repeatedly breaking up tasks into smaller and smaller subtasks until they have primitive, executable tasks. A primary reason behind HTN's success is that its task networks capture useful procedural control knowledge described by a decomposition of subtasks (Nau, 2003). Such control knowledge can significantly reduce the search space for a plan thus making it a fast planner and ideal to use in restricted processor environments like games.

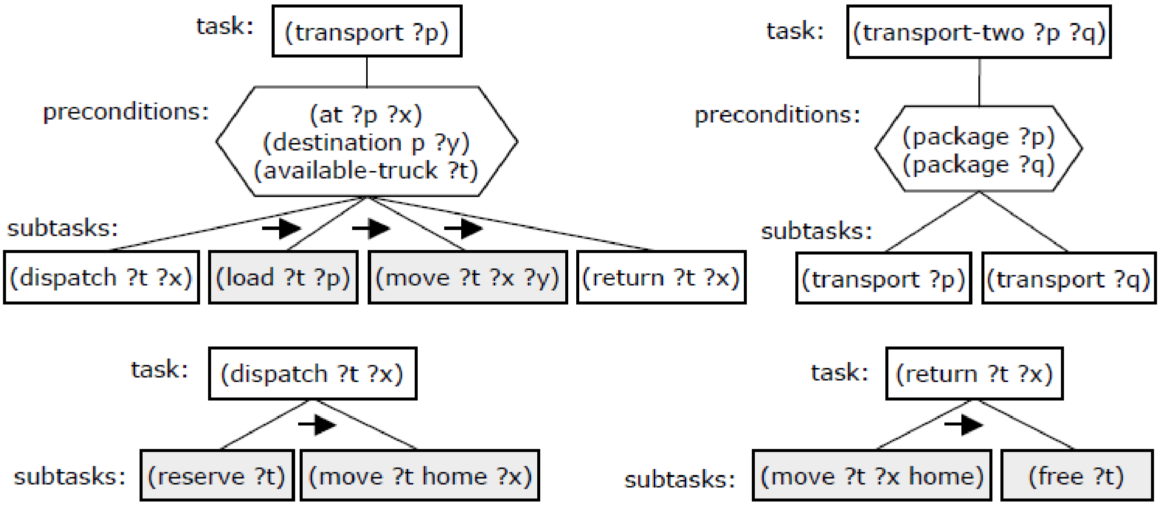


Figure 13 - an HTN example

The example above shows the transport procedure of two packages. First we check the preconditions for transporting both packages. If this fails we can quickly move on to the next task without looking more into the tree. If the transport can take place we look at the task of transporting a package. If these preconditions fail we move on, else we need to perform the subtasks dispatch, load, move and return in this order. We know we can perform these tasks because the preconditions are valid.

### *PDDL*

Planning domain definition language is another way of describing situations and come up with an optimal solution to reach a goal state. It was created in 1998 by Drew McDermott for the first International Planning Competition (IPC).[5]

The first version of PDDL used Boolean operators only, like STRIPS. The idea is to calculate every possible course of actions to reach the goal and then show the most efficient one (for example, a minimum of steps).

Parts of a PDDL planning task:

- objects/types: things in the world that interest us
- predicates: properties of objects that we are interested in; can be true or false
- actions/operators: ways of changing the state of the world
- initial state: the state of the world that we start in
- goal specification: things that we want to be true.

The first three parts define the PDDL domain. Different scenarios can use this domain. It only describes the object types, their properties and possible interactions. To solve multiple scenarios that contain the same objects we can use the same domain specification. The definition of a scenario exists of a domain, an initial state and a goal. The objects that interact are defined within the initial state specification.

[5] http://ipc.informatik.uni-freiburg.de/PddlExtension

```
(define (domain BLOCKS)
  (:requirements :strips :typing)
  (:types block)
  (:predicates (on ?x - block ?y - block)
               (ontable ?x - block)
               (clear ?x - block)
               (handempty)
               (holding ?x - block)
               )

  (:action pick-up
           :parameters (?x - block)
           :precondition (and (clear ?x) (ontable ?x) (handempty))
           :effect
           (and (not (ontable ?x))
                (not (clear ?x))
                (not (handempty))
                (holding ?x)))

  (:action put-down
           :parameters (?x - block)
           :precondition (holding ?x)
           :effect
           (and (not (holding ?x))
                (clear ?x)
                (handempty)
                (ontable ?x)))
  (:action stack
           :parameters (?x - block ?y - block)
           :precondition (and (holding ?x) (clear ?y))
           :effect
           (and (not (holding ?x))
                (not (clear ?y))
                (clear ?x)
                (handempty)
                (on ?x ?y)))
  (:action unstack
           :parameters (?x - block ?y - block)
           :precondition (and (on ?x ?y) (clear ?x) (handempty))
           :effect
           (and (holding ?x)
                (clear ?y)
                (not (clear ?x))
                (not (handempty))
                (not (on ?x ?y)))))
```

**Snippet 4 – PDDL Domain definition example**

Showing in Snippet 4 only one object type is defined: block. Predicates are true/false conditions. PDDL assumes all predicates to be false by default. Actions have parameters that locally define block objects (defined using a question mark followed by a variable name). We can use them to test preconditions or to show which values will change when the action occurs.

```
(define (problem BLOCKS-4-0)

(:domain BLOCKS)

(:objects D B A C - block)

(:INIT  (CLEAR C)
        (CLEAR A)
        (CLEAR B)
        (CLEAR D)
        (ONTABLE C)
        (ONTABLE A)
        (ONTABLE B)
        (ONTABLE D)
        (HANDEMPTY))

(:goal (AND (ON D C) (ON C B) (ON B A)))
)
```

**Snippet 5 - PDDL Scenario definition example**

In Snippet 5 the “scenario” is defined. Multiple object instances are now defined and assigned to a variable (in the example the letters A,B,C,D). The initialization part defines predicates that are true; the system assumes all others to be false. Using the specified goal, a PDDL interpreter can search for a solution which will consist of a series of actions that need to be performed to reach the goal state.

In PDDL version 2.1 fluents were added. These are numerical values that can be compared using logical operators. This also broadened the perspective to allow durational actions. An action could start at time T and a calculative function would determine the time before the action is complete. They included the choice to interrupt these actions when preconditions fail (Olaya, 2007). This expanded the possibilities for PDDL but also created much larger problems with many more states. This made calculations more intensive and it would take much longer to find the optimal solution. It is unsuitable to use this version for real-time decision making.

PDDL version 3 (latest version is 3.1 Developed by Alfonso Gerevini and Derek Long for IPC5 (ICAPS 06)) added goal preferences. This way we can improve a plan by pointing out what states we have to avoid or take. This concentrates more on quality then the result. If we select preferences carefully, this also improves the time it takes to find the first valid solution.

### 2.3.3 Conclusion planning languages

The first feature of planning languages we can to use for GSDL is to separate the domain information from the scenario itself. Multiple scenarios can reuse the same domain definition. This helps us to decrease **P6** (sections 1.2) by reusing existing information instead of duplicating information. It also supports the modularity of **R5** from section 1.3. Using a domain description, the programmer can create initial information for the domain. This way he can assure the domain expert will add certain values or include the restrictions of the simulator. This supports **R1** and **R3**.

The second feature we use is the preconditions. Using these to find out when a specific action can happen we can assure completeness of information (**R2**). We can provide warnings when information is missing or when actions can never occur. We can support playing back a scenario using a GSDL simulator and warn the domain expert when he wants to perform an action which is not possible. Then we can provide the domain expert with information about what preconditions enable the action.

The final features we will use are the predicates and fluents. Using these we can insert information like the number of people a fire truck can transport or whether a person has a fire extinguisher. This will support **R2** and will allow us to create advanced preconditions to support the extendibility of **R3**. Several numbers will be the internal values that are unknown by the domain expert.

| | R1 | R2 | R3 | R4 | R5 |
|---|---|---|---|---|---|
| **Strips** | | X | | | X |
| **HTN** | | X | X | | X |
| **PDDL** | | X | X | | X |

**Table 3 – Planning Language versus Requirement table**

Table 3 shows which requirements are met by the planning languages.

## 2.4 Conclusion background information

We researched three associated areas for scenario description.

1. Movies and games.
2. Existing scenario description languages.
3. Planning languages.

Each area coped with a similar knowledge gap in a different way, each with its advantages and disadvantages. None satisfied to all of our requirements, we picked the following features from these areas that help toward solving our problem:

- allow the programmer to alter the domain expert's software program to enforce adding valid entries (2.1.3)
- allow the domain expert to add textual information to clarify things if needed (2.1.4)
- use a well-known object oriented script language to define the scenario (2.2.2)
- allow description of behavior (2.2.3)
- separate the domain definition from the scenarios (2.3.2)
- add actions with preconditions like planning languages to be able to define an extensive scenario without describing each state individually (2.3.2)
- allow the programmer to define the boundary of the simulator (2.3.3).

We will combine these features to mold a scenario description language.

Using knowledge from this background information we can satisfy many requirements. However, meeting **R4** is a problem for all language based solutions. We can create a base for a general solution, but need to deal with intuitive and extensive usability in another way. Our solution, a scenario description tool which supports the user with a graphical user interface, is described in section 4.

# 3 Generic Scenario Description

## 3.1 Introduction

According to the dictionary a scenario is *"An outline or model of an expected or supposed sequence of events"*. In our case we decided that this model can be described in a structured piece of plain text. We call this our Generic Scenario Description Language (GSDL). This language should be writable by the domain expert and readable by the programmer. To help the domain expert by writing a code language we should develop the language in a way that a graphical user interface can interpret and export this code. We develop our language conform our requirements described in section 1.3.

In our scenario the sequence of events is not fixed. User interaction can alter the sequence and the outcome of the scenario. We should describe all the possibilities and boundaries of the supposed sequence of events to define the whole scenario. During the development, the domain expert should be able to test these boundaries with the GSDL simulator described in section 3.5.

The following section describes the environment in which we will use our tools. The workflow will affect some of the features of GSDL.

Then we will describe features and decisions in the next sections based on an example scenario. We will look at the sequence of events that takes place and how we can use our language to define this scenario.

## 3.2 Practical Environment

The environment in which we will use the description language is important for the structure of the language.

Most existing techniques use the following method: first the programmer creates some basic rules and objects for the expert to use. Like the FGD, EDF, MSDL-Schema or the domain definition of PDDL (section 2). Then the domain expert learns how to use these rules and objects and creates a scenario. Flaws are reported to the programmer (note that not all existing methods can deal with these flaws).

We borrow this idea from these existing techniques. We divide the scenario description in two parts; the first part to define the domain and abilities of the simulator and the second part to define the scenario itself. This supports **R3** and **R5**. The programmer can start by creating a set of default rules and objects to define the possibilities of the simulator defining the template of the scenarios. Figure 14 depicts the suggested workflow. Both parties can work simultaneously which allows the process to finish faster and thus offering a possible solution to our main problem.

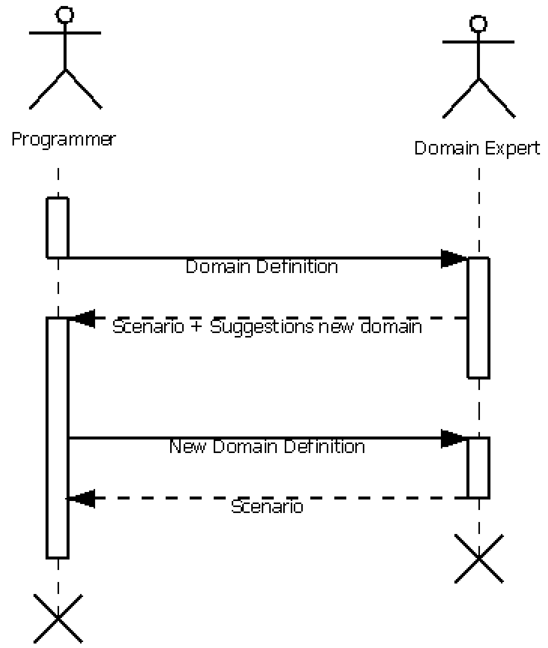


**Figure 14 - Sequence Diagram Workflow of creating a domain definition**

The first step is for the programmer to create a domain definition. This defines the possibilities of the simulator. If the simulator does not yet exist, the main ideas of the programmer about what features he is planning to implement can be described here. It serves as an API to the domain expert. The scenario description tool of the domain expert uses this domain definition; providing him with default objects and rules to create a first version of a scenario. The description tool is also responsible for a valid output of a GSDL defined scenario to prevent the domain expert from solving parse errors. Section 4 describes this scenario description tool.

If the domain definition is not enough to support the entire scenario, the tool the domain expert uses can expand the domain if necessary by adding more actions or objects. Here is the difference with a normal API, the domain expert can make changes to the domain definition. This will save time (**P0**) because the domain expert can continue working on the scenario, even if there was no support for the initial scenario.

The programmer can compare both versions of the domain definition and decide whether the changes are possible or if the domain definition needs to be defined in a different way. The programmer will know the intents of the domain expert and can effectively look for an alternate solution if the supplied version is impossible to implement directly.

Meanwhile the domain expert can work on completing the scenario.

The script will allow performing tests on the script itself using a GSDL simulator. Therefore allowing the domain expert to execute some simple tests to make sure the simulation will do what it is supposed to do. This will help the programmer understand what the goals of the simulation are. We can use these tests to “ask” specific questions to the domain expert. The programmer just adds a test and asks the domain expert if this is the correct result. If it’s not, the domain expert can try to change the script until all tests work as wanted or, if the task is too difficult, the programmer can do this himself since he now knows what the results must be. We discuss this idea in section 6.3.3.

## 3.3 Development

We describe the development of the generic scenario description language (GSDL). The GSDL language is used by the GSDL simulator (section 3.5) and the Scenario Description Tool (described in section 4). The language should be interpretable by the programmer and possibly extendable to be used directly with other simulators.

We start using an object oriented script language (Java) to describe a simple scenario. We use the benefits of the background information to optimize the usability for scenario description. We use this simple scenario to describe the generic architecture of a simulation in section 3.3.5. Finally we use this information and the Java code to strip the unused Java features which will leave a concise description language for scenarios. We make sure we can convert this language back to Java to keep the feedback features. This conversion process is described in section 3.4.

In section 3.5 we start building a simple discrete GSDL simulator that can simulate the objects and possible actions described in a GSDL scenario. This is important for the domain expert to be able to test the scenario providing feedback to make sure it works as it is supposed to. We need it to provide information about the state and show what actions are available.

Within these sections we present snippets of code that help explain the text. Appendix 0 describes the complete and uncensored code.

In section 3.6 we put our language to the test, discussing benefits and drawbacks and comparing our language to some comparable ideas.

### 3.3.1 Describing a simple situation

We want to describe the following simple scenario:

1. A fire starts at a specific location.
2. The fire department sends a fire truck with 2 firefighters to the location.
3. The firefighters deploy at the location and extinguish the fire together.

We assume the virtual environment of this scenario is already present in some way. This can be a list of locations, but also a 3-dimensional environment. The final language should be able to cope with both kinds. Depending on the virtual environment the GUI for the domain expert will be different; we discuss this in section 4. For our example we assume that a location is defined by a name and each object is able to be at a maximum of one location at the time.

We chose Java as a base for our description language. We use the advantages from the object oriented approach (section 2.4) and it is a well-known language for programmers. Thanks to the use of a virtual machine it can also run on multiple platforms. The next sections describe the implementation of the rest of the features based on a sample scenario.

The first step we need to take is to describe objects that persist within the simulation environment. We describe the main objects in the domain definition. Then we can introduce multiple instances of them in the scenario itself.

From our simple scenario we extract the following objects:

- a specific location
- a fire
- a fire department
- a fire truck
- a fireman.

We want each object to have some generic features, so we create a BaseObject from which each object inherits generic features. We add a name for each object to identify the object.

```
public abstract class BaseObject {
    private final String name;

    public BaseObject(String name) {
        this.name = name;
        this.init();
    }

    protected void init() {
    }

    public final getName() {
        return this.name;
    }
}
```

To create a person we extend the BaseObject. The same can be done for a fire.

```
public class Person extends BaseObject {
        public Person(String name) {
                super(name);
        }
}
```

A fireman is an extension to the object person. So we use the Java inheritance to define fireman. To create a fireman we extend the person object.

```
public class Fireman extends Person {
        public Fireman(String name) {
                super(name);
        }
}
```

This way we can add features to a person which each fireman would automatically inherit. For the fire truck and a vehicle we can do the same. This is straightforward to Java programmers. This modular approach suits **R5** by adding modularity to the scenarios. Once we create a set of base objects, we can reuse them for multiple scenarios. These objects can easily be extended to custom objects with similar features.

The actions we will create can allow one type of object to perform it. If we create an action that can be performed by a "person", each "fireman" will also have the ability to perform that action since they also classify as a "person".

The BaseObject class defines how we use our language. The simulation programmer will not be able to change this class. Therefore we put this class in a package "base". The other classes reside in the domain definition so we put these in a package "domain". The programmer will create or edit these classes and later possibly the domain expert will edit the domain as well. The use of a domain package matches to the requirement **R3**, changes to the domain could mean the programmer should add these features the simulator.

Each object is at maximum at one location at the time. So we define the location for each object in the BaseObject class. To be able to use different location properties for different simulators we define a location object that can be expanded later. For now we assume a location is defined by a unique name.

Objects often depend on other objects for its location. For example, a mobile phone might be carried by another object or a nozzle can be attached to a hose. Therefore we introduced the possibility to attach objects to other objects. Each object can be attached to one other object and will share the location of that object.

To allow user interaction we need something to define where a user can exert influence on. The trainee can make decisions within the scenario, but this can also be the teacher or even a piece of software. We define a controller object for each BaseObject. This controller object should receive all perception events for this object and it should be able to tell the object when to perform certain actions.

Appendix 0 describes the final version of our BaseObject.

### 3.3.2 Predicates, Functions and Messages

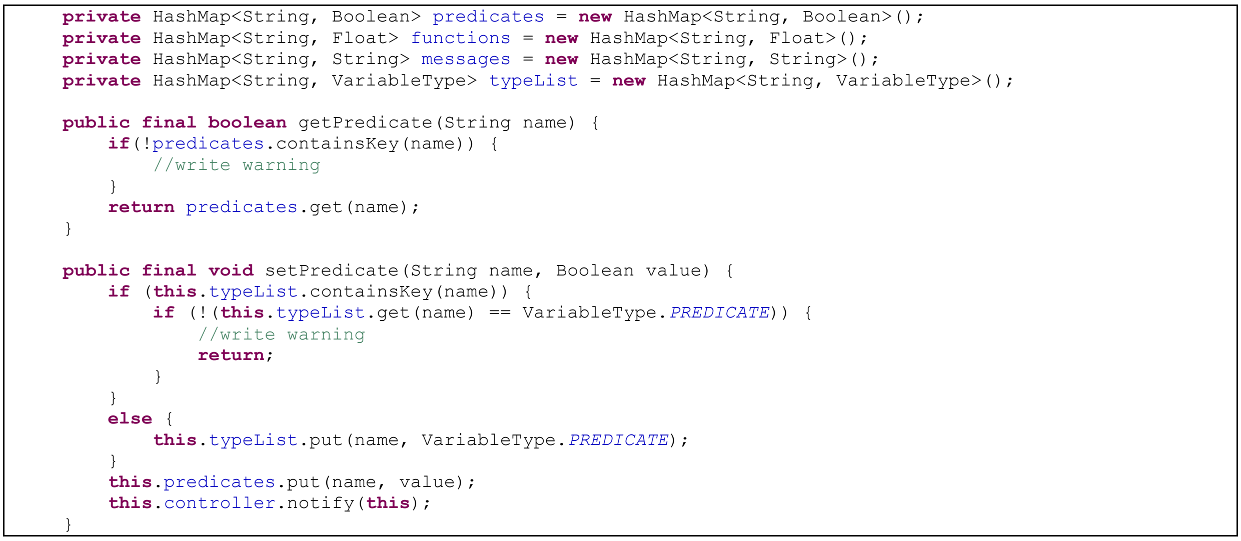
From the PDDL language we learned that we can use predicates and functions to define the characteristics of objects. In our language we can define these in the object class within the domain specification. We define a set-method to be able to add predicates to an object. We use a string as identifier. Using this string we can request the suiting value.

We found that we need to add messages for use in simulations. Next to planning a scenario we also need to provide textual information to the trainee. So we need a way to define text based values for our objects. We add them just like we do with predicates and functions.

We use the following types of variables in our language:

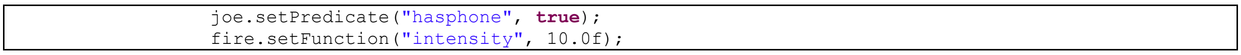
- predicates (Boolean) – setting true or false conditions
- functions (float) – adding numeric values and the capacity to use them for calculations
- messages (string) – saving textual information or to present users with a helpful message.

We add the following code to the BaseObject class for the variables. We decided that we can use one name for a specific parameter. This aids the intelligibility (**R2**) and allows us to find out what (type of) parameter belongs to any given name which aids the lookup procedure.



```
    private HashMap<String, Boolean> predicates = new HashMap<String, Boolean>();
    private HashMap<String, Float> functions = new HashMap<String, Float>();
    private HashMap<String, String> messages = new HashMap<String, String>();
    private HashMap<String, VariableType> typeList = new HashMap<String, VariableType>();

    public final boolean getPredicate(String name) {
        if(!predicates.containsKey(name)) {
            //write warning
        }
        return predicates.get(name);
    }

    public final void setPredicate(String name, Boolean value) {
        if (this.typeList.containsKey(name)) {
            if (!(this.typeList.get(name) == VariableType.PREDICATE)) {
                //write warning
                return;
            }
        }
        else {
            this.typeList.put(name, VariableType.PREDICATE);
        }
        this.predicates.put(name, value);
        this.controller.notify(this);
    }
```

An object is now represented by a name, optional predicates, functions and messages. We can now add the predicate "hasphone", for example, to show the object has a phone. We can then test whether this predicate is true before allowing the person to call someone (or be called). Using functions we can add numeric values to objects. For example, intensity rate to the fire.



```
                    joe.setPredicate("hasphone", true);
                    fire.setFunction("intensity", 10.0f);
```

Using a string as identifier allows a flexible use of variables. Using this way of setting variables, we do not need to be predefine them and we can add them on the fly.

This also allows the extensive usability from requirement **R4**. The language does not limit the domain expert because he can express every feature of an object if necessary. The predicates can be compared using logical operators and the functions can be used for the mathematical features of the simulation.

### 3.3.3 Relations

We found that it can be useful to describe relations between objects. Using the same idea from the predicates and functions we can define relations between specific objects. Again we use a string as identifier.

```
    private HashMap<String, Vector<BaseObject>> relations = new HashMap<String, Vector<BaseObject>>();

    public final void setRelation(String name, BaseObject object) {
        if (this.typeList.containsKey(name)) {
            if (!(this.typeList.get(name) == VariableType.RELATION)) {
                //write warning
                return;
            }
        }
        else {
            this.typeList.put(name, VariableType.RELATION);
        }
        if (!this.relations.containsKey(name)) {
            Vector<BaseObject> v = new Vector<BaseObject>();
            v.addElement(object);
            this.relations.put(name, v);
        }
        else {
            this.relations.get(name).addElement(object);
        }
        this.controller.notify(this);
    }

    public final void removeRelationWith(String name, BaseObject object) {
        if (!this.relations.containsKey(name)) {
            return;
        }
        if (this.relations.get(name).contains(object)) {
            this.relations.get(name).remove(object);
            this.controller.notify(this);
        }
    }

    public final boolean hasRelationWith(String name, BaseObject object) {
        if (!this.relations.containsKey(name)) {
            return false;
        }
        for (BaseObject o : this.relations.get(name)) {
            if (object == null || o.equals(object)) {
                return true;
            }
        }
        return false;
    }
```

We can now define that a fireman is sitting inside a fire truck, also the other way around we can set the relation describing the fire truck contains a fireman.

```
    f1.setRelation("inside", truck);
    truck.setRelation("contains", f1);
```

Before events may happen we can test if these features of objects are true. We could also provide a warning or explanation why an event cannot take place (for example, truck cannot drive because there is no fireman inside the truck).

During the creation of GSDL we found that we can replace relations by predicates using an objectname within the identifier. We could replace the code above by the following:

```
    f1.setPredicate("inside " + truck.getName(), true);
    truck.setPredicate("contains " + f1.getName(), true);
```

Due time constraints we did not change this. But a description how to do this can be found in section 6.3.2.

### 3.3.4 Events and Actions

Using the previous definitions we can define a state of the simulation. We now need to define something that allows changing the state of the simulator. We use event and action classes to define a state change.

An event exists of two main features: a precondition and the preformed changes if the event takes place. These features exist in STRIPS as well as PDDL and HTN (see section 2.3.2) and proved to be usable for scenario description. When preconditions of an event become valid it performs immediately. This idea from planning languages allows logical tests to decide when a state-change can occur. Using this theory from the planning languages allows us to add conditional events (supporting **R1**).

An action looks like an event except an action cannot perform by itself; an object is needed to perform the action. Each action has exactly one object that can initiate that action. In a scenario there are many possible actions defined but only a few will be in an enabled state. When the preconditions of an action become valid the action will become enabled and the corresponding object can choose to perform the action.

The controller of an object should be able to list all the possible actions, preferably also the disabled actions and the reason the action is disabled in the GSDL simulator. This way you can learn from the scenario… For example, *"I cannot move the truck because there is no fireman inside"*. This answers to requirement **R2** by supporting feedback when performing a test.

To add events into our code we create a BaseEvent class in our base package. This class should assure the presence of the two features of an event.

```java
package base;

public abstract class BaseEvent {

    public abstract boolean enabled();
    public abstract void perform();

}
```

An action will define the first object in the constructor as the initiator of the action. This will result in adding this action to the actionlist of this object.

One feature that is present in every simulator but not specifically defined is the time. To allow events to happen on a predefined time we add the Simulator class. This class contains simulator specific information, for now just the time. Below an example how we can trigger an event on a specific time.

```
package domain.events;

import base.*;

public class MessageEvent extends BaseEvent {

    public MessageEvent() {
    }

    @Override
    public boolean enabled() {
    if (Simulator.getTime() > 10f) return true;
        return false;
    }

    @Override
    public void perform() {
    }

}
```

An interesting problem shows up using events; if two events are enabled at the same time, which one will execute? One could argue that both can execute, but this complicates when one event disables the other. For now we assume the simulation becomes nondeterministic when we use events that can happen at the same time and that only one event can happen at one time. We address this problem in section 3.6.2.

We can create tests to see if specific events can happen. For example, if we want to test if the incoming call can also occur at the 9th timeslot, we can invoke the IncomingCall.enabled() function which would return false. To fully understand why this is not possible and which values need to change before we can perform this event we need a textual representation of the preconditions.

For example, EnterVehicle action:

```
public class EnterVehicle extends BaseAction {

  private Person person;
  private Vehicle vehicle;

  public EnterVehicle(Person person, Vehicle vehicle) {
    this.person = person;
    this.vehicle = vehicle;

  }

  @Override
  Public boolean enabled() {
    if (!(person.getLocation().equals(vehicle.getLocation())))return false;
    if (!(vehicle.getValue("occupants") < vehicle.getValue("seats"))) return false;
    if (!(!person.hasRelationWith("inside", vehicle))) return false;
    if (!(!person.isAttached()))return false;
    return true;
  }

  @Override
  public void perform() {
      person.setRelation("inside", vehicle);
      vehicle.setFunction("occupants", vehicle.getFunction("occupants") + 1f);
      person.attachToObject(vehicle);
  }
}
```

This action enables when both the person and the vehicle are at the same location and the person is not already inside another vehicle. If this action is disabled we do not know whether the location causes this or if this person is already inside another vehicle (or both). We can add an extra method that provides us with the extra information:

```
@Override
public String enabledStatus() {
  String s = "";
       s += (person.getLocation().equals(vehicle.getLocation()) ? "" : "(!) ") + person.getName() + " has 
the same location as " + vehicle.getName() + "\n";
       s += (vehicle.getValue("occupants") < vehicle.getValue("seats") ? "" : "(!) ") + 
"vehicle.getValue(\"occupants\") < vehicle.getValue(\"seats\")\n";
       s += (!person.hasRelationWith("inside", vehicle) ? "" : "(!) ") + person.getName() + " has no 
relation \"inside\" with " + vehicle.getName() + "\n";
       s += (!person.isAttached() ? "" : "(!) ") + person.getName() + " is not attached\n";
  return s;
}

@Override
public String toString() {
  return "EnterVehicle"+ " (" + this.person.getName() + ")"+ " (" + this.vehicle.getName() + ")";
}
```

We can draw this piece of code automatically from the enabled function. It is not necessary for a domain expert to write down this code. This is an implementation problem.

The most difficult code the domain expert should insert is the following:

```
    if (!(person.getLocation().equals(vehicle.getLocation())))return false;
    if (!(vehicle.getValue("occupants") < vehicle.getValue("seats"))) return false;
    if (!(!person.hasRelationWith("inside", vehicle))) return false;
    if (!(!person.isAttached()))return false;
```

This is simplified, but still hard for a non programmer to understand.

If you would ask someone when a person can enter a vehicle, the answer would be: "*When the person and the vehicle are at the same location*". This could translate to person's location equals vehicles location. We have to make this translation as easy as possible for the domain expert.

The second interesting part is the perform function. We want to define what happens if we perform an action.

The first information our system needs to describe an action is the initiator. From a list of defined actions the domain expert chooses what object (type) is able to initiate this action. We can select second optional other objects from which this action needs information or on which this action performs change. For the example above it is not difficult to realize that a person is the initiator of EnterVehicle and that we need an object Vehicle to perform this action with.

Finally when we create a new action we possibly introduce some new functions or predicates. These might put restrictions on existing actions. If so, this is the time to present the domain expert with this fact. For example, when we attach this object, actions like "*enterVehicle*" should use this as a restriction.

### 3.3.5 Architecture

We depict the architecture of the example used above in the UML Class diagram below. The base objects contain the necessary functions to describe a scenario. This is not part of the scenario description itself; it is the tool that happens to be written in the same language as the description.

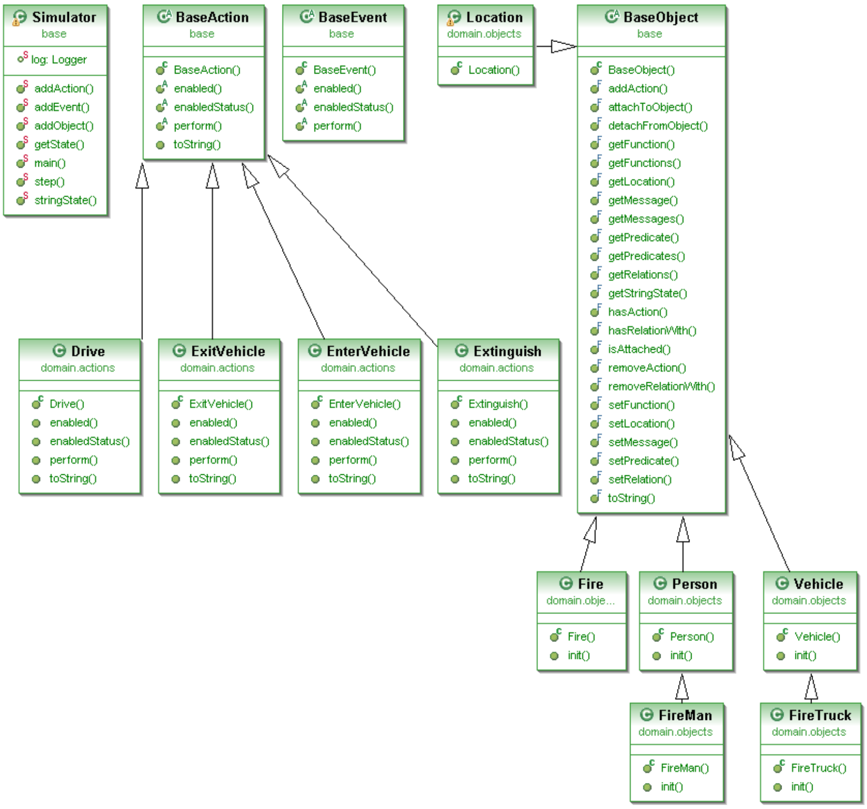


**Figure 15 - Example Architecture**

The image shows the objects that we can instantiate. This is our domain definition. We define the instantiated objects separately in a scenario. The scenario will use these objects from our domain definition.

We can use one domain definition to create multiple scenarios.

### Object

An object represents an entity within the simulation. We can extend it to inherit all functions that its parent object has. It will automatically be able to perform all the actions of its parent. The domain definition of an object describes a default state of any object that we create using this class. The important fields are:

- objectname (Name of the object type)
- parentname (Optional, name of the parent of this object type)
- predicates (Optional, boolean values, true or false)
- functions (Optional, numeric values)
- messages (Optional, textual values).

We do not define the following fields in the domain definition since they are different for each object and should be defined in the scenario part.

- controller (Default = none, is AI- or user-controlled)
- relations (Lists of objects that relate to this object)
- actions (The actions this object can perform by itself).

The only compulsory field within the domain definition is the objectname. We define a new object using the following syntax:

```
object <objectname>
```

Or including a parent using the "extends" syntax:

```
object <objectname> extends <parentname>
```

We translate this automatically to the following Java syntax:

```
package domain.objects;

public class <objectname> extends <parentname> {

        public <objectname>(String name) {
                super(name);
        }

}
```

We can use this in our GSDL simulator. Notice that we need a name to construct any object.

To save time we want to define some default values for a specific object and make use of the advantages of object oriented programming. Defining default values will save time for the domain expert because he only needs to supply the differences to the default values. Therefore we introduce the init() function that we call on creation of the object or any children of this object. We can initialize predicates, functions and messages. In the pseudo code below we use square brackets to mark optional code and inequality signs to mark variable values. The curly braces are part of the code.

```
object <objectname> [extends <parent>] {
    [predicate <predicatename> = <true/false>;]
    [function <functionname> = <numericvalue>;]
    [message <messagename> = <string>;]
}
```

**Snippet 6 - Pseudo code Object Definition**

For Example:

```
object Firetruck extends Vehicle {
    predicate "isDriving" = false;
    function "waterVolume" = 1500;
    function "seats" = 3;
    message "vehicleId" = "V001";
}
```

This would translate into the following Java code:

```
package domain.objects;

public class Firetruck extends Vehicle {

    public Firetruck(String name) {
        super(name);
    }

    public void init() {
        super.init();
        this.setPredicate("isDriving", false);
        this.setFunction("waterVolume", 1500f);
        this.setFunction("seats", 3f);
        this.setMessage("vehicleId", "V001");
    }
}
```

By introducing the init() method we make sure that we call this method for each created instance. We can do this in the BaseObject constructor. We cannot create default relations or controllers since these are different for each instantiated object and need to be defined in the scenario definition.

If there is extra information we can add about a certain value, it is always possible to add a commentary line, starting with a double slash forward. We want to keep this option open because there might be pieces of code where it is still too difficult for the domain expert to create the code himself and wants to leave a helpful remark to the programmer.

### Event

An event is something that can happen within the simulation. If its conditions apply it will perform changes to objects.

- eventname
- prerequisites (Set in constructor, defines what objects we need to perform this event)
- enabled (Logic function to tell whether this event can take place)
- perform (Description about what values this event changes each step).

### Action

An action defines the capacity of an object to make changes to its environment. If its conditions apply it can perform if an object triggers it to do so.

- actionname
- initiator (Set in constructor, defines what objects can initiate this action)
- prerequisites (Set in constructor, defines what objects we need to perform this action)
- enabled (Indicates the conditions that we must meet before this action can perform)
- perform (Generates an enabled event which describes what will happen).

For each of these features we define a piece of code that contains all the necessary information needed for us to create valid Java-code. This is presented in table 4.

| Feature | Info Needed | Target Java code |
|---|---|---|
| Actionname/ Eventname | Name | `public <ActionName>() {}` |
| Initiator | ObjectType, LocalObjectName | `private <ObjectType> <LocalObjectName>;`<br>`public <ActionName>(<LocalObjectName>) {`<br>`   this.initiator = <LocalObjectName>;`<br>`}` |
| Prerequisites | ObjectType, LocalObjectName | `private <ObjectType> <LocalObjectName>;`<br>`public <ActionName>(<LocalObjectName>) { }` |
| Enabled | Logic tests | `public boolean enabled() {`<br>`  [if(!(<Logic test>)) return false;]*`<br>`  return true;`<br>`}` |
| Perform | Variable Assignments: LocalObjectName, ValueType, ValueName, Value | `public void perform() {`<br>` <LocalObjectName>.set<ValueType>(<ValueName>,`<br>`<Value>);`<br>`}` |

Table 4 – Required information table

We can also reuse the logic tests information to create enabledStatus() code described in section 3.4.3. We now know what information we need to define in GSDL. How we define this is described in the following section.

## 3.4 Generic Scenario Description Language

In this section we describe how we develop a structured language (GSDL) to define a scenario and how we convert this simple language to more complicated useful Java code. This code is used in the GSDL simulator described in section 3.5. We use the information from our findings in the previous sections and combine this be able to create an efficient description of a scenario. Appendix 8.1 shows the complete EBNF[6] notation of GSDL.

We use the ANTLR (Parr, 2007) tool to assure consistency of our language. ANTLR stands for ANother Tool for Language Recognition; it is a language tool that provides a basic structure for constructing recognizers, interpreters, compilers, and translators from grammatical descriptions containing actions in various target languages, including Java. Appendix 8.4 shows the complete grammar file for GSDL.

[6] Extended Backus–Naur Form

### 3.4.1 Lexer

The idea is to create a lexer (or “scanner”) to remove whitespaces and comments. Then the lexer defines the useful tokens. Using these tokens we can use a parser to interpret the code and parse this to Java code. This will produce parse errors if the code is not consistent and it will not be able to detect ambiguous code thus ensuring unambiguous code.

We display the lexer at the bottom of appendix 8.4. We define identifiers as a word starting with an uppercase character, variables as a word starting with a lowercase character and strings as characters between double quotation marks (”) this is comparable with the Java syntax.

### 3.4.2 Object Declaration

In the previous section we depicted the 5 important elements of the object declaration. We combine these in our parser. Appendix 8.1 shows the EBNF notation of the object declaration. ANTLR creates the following visual railroad view.

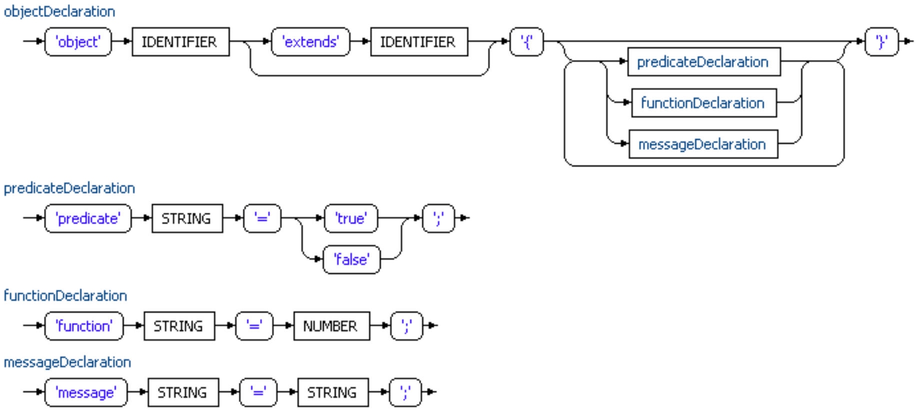


**Figure 16 - Railroad view of object declaration**

The first identifier is the name of the object. The second is the optional name of the parent object. Then we define predicates and functions using a keyword at the start and a semi-colon at the end. This is all the information we might need to create an object for our Domain Definition.

### 3.4.3 Action and Event Declaration

Actions and events are similar. The only difference is that an action has an initiator and an event has not. The railroad view of an action shows what how we define an action.

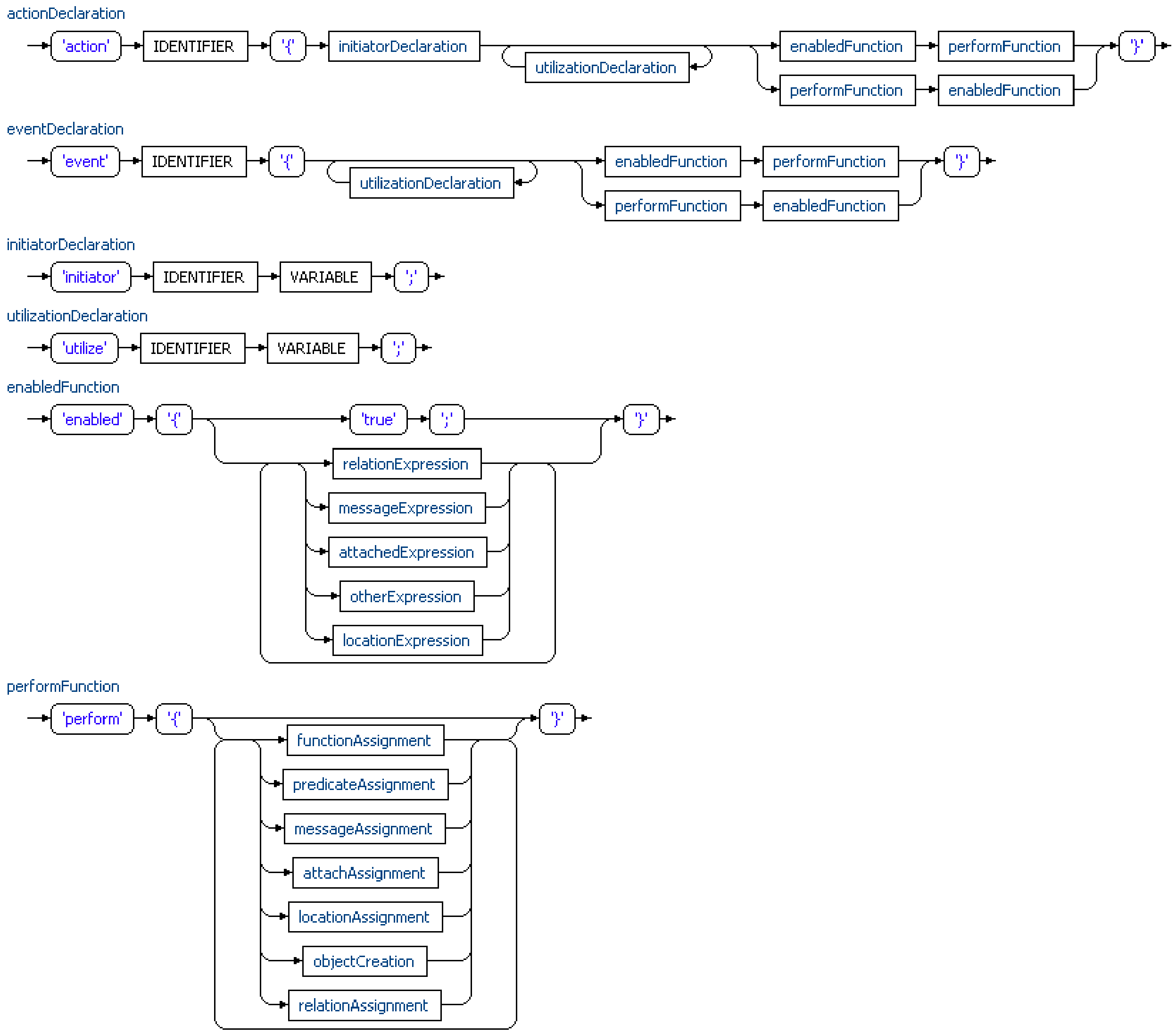


**Figure 17 - Railroad view of actions and events**

At declaring the initiator and the utilization, we match objects (defined by the identifier) to a variable. We can use these variables in the expressions in the enabled and perform functions. We describe an example of the EnterVehicle action below. The expressions and assignments are straightforward, the complete list of railroad views can be found in appendix 8.2.

```
action EnterVehicle {
  initiator Person person;
  utilize Vehicle vehicle;

  enabled {
    person location equals vehicle location;
    vehicle "occupants" < vehicle "seats";
    !person hasrelation "inside" vehicle;
    !person is attached;
  }

  perform {
    relation make person "inside" vehicle;
    function vehicle "occupants" = vehicle "occupants" + 1;
    attach person to vehicle;
  }
}
```

**Snippet 7 - EnterVehicle Action in GSDL**

This action is enabled if the following 4 predicates are true:

1. The location of person and vehicle is the same.
2. The number of occupants is smaller than the number of seats.
3. The person trying to enter does not have a relation inside with the vehicle.
4. The person is not attached to another object.

If this action performs the following changes to the environment will take place:

1. A relation “inside” is created for the person who invoked this action with the vehicle.
2. The function “occupants” of vehicle is increased by one.
3. The person becomes attached to the vehicle.

### 3.4.4 Scenario declaration

Using a domain definition we can define a scenario by defining objects and the initial state of those objects. The scenario declaration looks like the perform function of actions and events.

```
scenario Scenario1 {
  Location fd = "Fire Department";
  Location home = "Home";

  FireMan f1;
  FireMan f2;
  FireTruck truck;
  Fire fire;

  location fire = home;
  predicate fire "burning" = true;

  location f1 = fd;
  location f2 = fd;
  location truck = fd;

}
```

**Snippet 8 – Scenario definition in GSDL**

First we need to declare the objects. We assign each object a variable name which we use later to change the default values of the objects like we do within a perform function (section 3.4.3). Finally we can define locations and initial variables or attachments if needed.

## 3.5 GSDL simulation testing

We can run our simulation in the GSDL simulator using the Java output of the GSDL parser. We run the scenario by creating the initial state where we instantiate all objects. At the end we can gather a list of all possible events and actions that can take place. For now we assume a discrete event simulation.

Advantages we can use thanks to our tool to assure completeness of the scenario are:

- creating a list of all the possible actions at the current state
- creating a list of actions for each controller
- creating validity tests.

The initial state is the collection of objects and their current predicates, functions and relations. To create a list of all objects we use the BaseObject class to add each instance to the GSDL simulator:

```
        public BaseObject(String name) {
                this.name = name;
                Simulator.addObject(this);
        }
```

Now we can think of the current state of an object using the following code.

```
    public final String getStringState() {
        String s = this.name + "\n";
        if (this.getLocation() != null) s += " location is " + this.getLocation().getName() + "\n";
        if (this.attached != null) s += " attached to " + this.attached.getName() + "\n";
        for (Map.Entry<String, Boolean> entry : this.predicates.entrySet()) {
            s += " predicate " + entry.getKey() + " is " + entry.getValue() + "\n";
        }
        for (Map.Entry<String, Float> entry : this.functions.entrySet()) {
            s += " function " + entry.getKey() + " is " + entry.getValue() + "\n";
        }
        for (Map.Entry<String, String> entry : this.messages.entrySet()) {
            s += " message " + entry.getKey() + " is " + entry.getValue() + "\n";
        }
        for (Map.Entry<String, Vector<BaseObject>> entry : this.relations.entrySet()) {
            for (BaseObject b : entry.getValue()) {
                s += " relation " + entry.getKey() + " " + b.getName() + "\n";
            }
        }
        return s;
    }
```

We can use the BaseAction constructor to add each action to a master list of actions. The number of objects and constructor types can change by editing the GSDL code; we need this to be independent. The following code manages this:

```
            for (Class<?> c : Simulator.getClasses("domain.actions")) {
                Simulator.checkAction(c);
            }
```

This takes all classes from the domain.actions package and sends them to the checkAction-method. Now we need to check what type of objects we need to construct this action. For each parameter we create a list of all objects that are the right type.

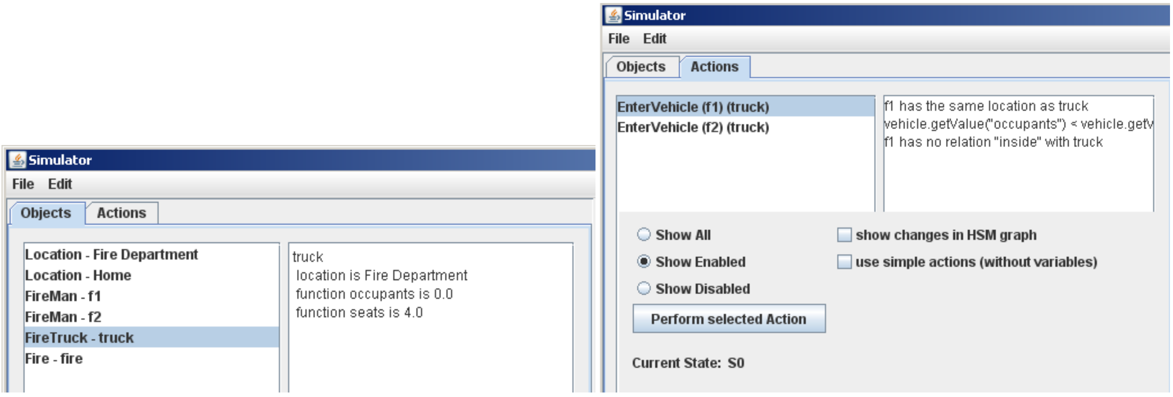
```java
private static void testActions(Constructor<?> c) {
//One vector containing X vectors with objects where X is the number of arguments needed to construct
action.
    Vector<Vector<Object>> parameters = new Vector<Vector<Object>>(c.getParameterTypes().length);
    int i = 0; // this int is reused multiple times in this method
    // Initialize an empty vector for each parameter.
    for (i = 0; i < c.getParameterTypes().length; i++) {
        parameters.add(i, new Vector<Object>());
    }
    //loop through all objects
    i = 0;
    for (BaseObject o : Simulator.objects) {
        //check if the classtype fits one of the classes
        i = 0;
        for (Class<?> cl : c.getParameterTypes()) {
            if (cl.isInstance(o)) {
                // add object to the associated list
                parameters.get(i).addElement(o);
            }
            i++;
        }
    }
    // If there is an empty vector this means this action cannot be created.
    for (Vector<Object> v : parameters) {
        if (v.isEmpty()) return;
    }
    // Use recursion to create all possible combinations of the objects in the vectors.
    recursiveCheck(new Vector<Object>(parameters.size()), parameters, c);
}
```

Finally we need to create all possible variations using all the current existing objects in the scenario. We use a recursive function to loop through all the possible combinations of objects to construct this action. We ignore actions where we would try to use the same object twice because we only need to initiate one object once in any constructor.

```java
private static void recursiveCheck(Vector<Object> v, Vector<Vector<Object>> parameters, Constructor<?> c) {
//Check if the vector with elements is filled. (iow: we have enough elements to create an action object)
// Check for duplicates: HashSet does not allow duplicates, skip all next iterations if a duplicate is
found.
    HashSet s = new HashSet(v);
    if (s.size() != v.size()) {
        return;
    }

    if (v.size() == parameters.size()) {
        //try to create our object
        try {
                // Create the action object using the parameters from our vector v.
                BaseAction action = (BaseAction) c.newInstance(v.toArray());
            }
        catch (Exception e) {
            e.printStackTrace();
        }
        //we are done! we return to the previous invocation.
        return;

    }
    //Loop through all possible parameters we can use as next argument
    for (Object o : parameters.get(v.size())) {
        // add the next element to the vector
        v.addElement(o);
        // pass the vector to check again
        Simulator.recursiveCheck(v, parameters, c);
        // when we return here, our element has been checked. remove it and try the next
        v.removeElementAt(v.size() - 1);
    }
}
```

At any given time we can check each action whether the enabled-method returns true. If this is the case, the action can perform. We could present the user with this list and let the user decide which action should take place. Then we can calculate the next state of the GSDL simulator.

We built a system that changes states using actions (labels), similar to Labeled Transition Systems (LTS, see section 3.6.4) which we can picture using state diagrams. We show an example of this visualization in section 0.



Location - Fire Department
Location - Home
FireMan - f1
FireMan - f2
FireTruck - truck
Fire - fire

truck
location is Fire Department
function occupants is 0.0
function seats is 4.0

EnterVehicle (f1) (truck)
EnterVehicle (f2) (truck)

f1 has the same location as truck
vehicle.getValue("occupants") < vehicle.getV
f1 has no relation "inside" with truck

○ Show All
● Show Enabled
○ Show Disabled
□ show changes in HSM graph
□ use simple actions (without variables)

Current State: S0

**Figure 18 – GSDL simulator examples**

The figure above shows the example we used in our GSDL simulator. We show more screenshots in section 0. This GSDL simulator will provide the domain expert with feedback and can offer help where needed. The domain expert can test if the scenario does what it is supposed to do. Also a programmer can understand what a given action does, even if the name is unclear.

Another feature is to create validity tests. One might want to assure that at some time in the simulation certain values are valid. For example, you might want to be sure that it is possible to extinguish the fire at some point. We can do this the same way as with planning languages. Using the planning language structure we can calculate every possible state of the GSDL scenario. The simulator can use an algorithm to find the fastest way to its goal, or just visit each state using graph theorems. We expand this idea in section 6.3.3.

## 3.6 Problems and Discussion

This section will discuss some of the problems we faced during the development of GSDL. Also we will make assumptions about the usability. We compare our solution to LTS and MAS to compare our GSDL simulator with similar solutions. We describe the conclusion about GSDL in section 3.7.

### 3.6.1 Usability

A GSDL script is still too difficult for the domain expert to understand. It does provide a solid platform for a user interface as we will discuss in section 4. This could result in an acceptable tool for the domain expert. The main question is if learning and using the GSDL tools will cost less time than intense communication. We claim that this is the case; we elaborate this claim in section 6. The tool supports the requirements to solve our final problem.

Another issue to find out is the usability of this language. We want to know whether it is scalable to describe each scenario that we can think of. We have shown it to work with simple scenarios. Large scenarios with big amounts of important objects can easily lose overview. Again the tool from section 4 can aid solving this problem.

Some other issues we stepped over for now still needs solving. The GSDL simulator time and location types may differ from the values we used here. Time might not be discrete and locations might be xyz-coordinates. We address this issue in section 6.3.2.

Another feature could be the easy implementation of durational actions (like in PDDL 2.1, section 2.3.2). For now we can do this for example by using a predicate "busy" that disables each action. Then we can use an event that will set this predicate to "false" when a timer has passed. This is a complicated solution, especially for a domain expert so a special function to support durational actions might help.

### 3.6.2 Concurrency

A few problems arise when testing a scenario. What if events can happen at the same time but also disable one another? This can also happen with actions that disable events and conversely. But we give priority to events above actions so this should not necessarily be a problem. We describe these situations using LTS notation (Magee & Kramer, 2006). Assume we have 3 events that can happen:

Now we define a situation where event_a blocks event_b.

We depict this state machine in Figure 19.

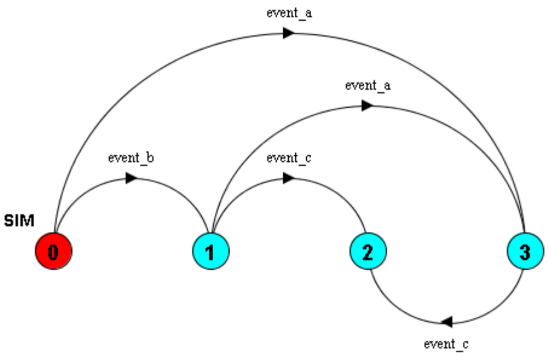


Figure 19 - State Machine Model of blocking events

The simulator has to make a choice in events, since this is not yet described in our language. Ideally one would say that the best sequence of events would be where we use the complete alphabet:

In the example this would be:

The problem is that the simulator does not by definition know what events block one another. We can make an assumption by checking if the events use shared resources. Another alternative is to explore each possible state and find out what events block each other. We can achieve this using state space exploration described in section 6.3.3.

### 3.6.3 Compared with MultiAgent Systems

Development of multiagent Systems (MAS) started late 1970 as the study of Distributed Artificial Intelligence (DAI). They define DAI by the following definition:

*DAI is the study, construction, and application of multiagent systems, that is, systems in which several interacting, intelligent agents pursue some set of goals or perform some set of tasks.*

Agents are autonomous, computational entities that perceive their environment through sensors and act on their environment through effectors (Weiss, 1999).

Our current GSDL simulator does not meet these requirements. Therefore we cannot say we have created a multiagent system. We created an environment for agents to run in. It is possible, using the controllers in our simulator, to create autonomous computational entities to control the objects.

According to Nwana, we can classify agents by the following three attributes: cooperation, ability to learn and autonomy (Hyacinth, 1996).

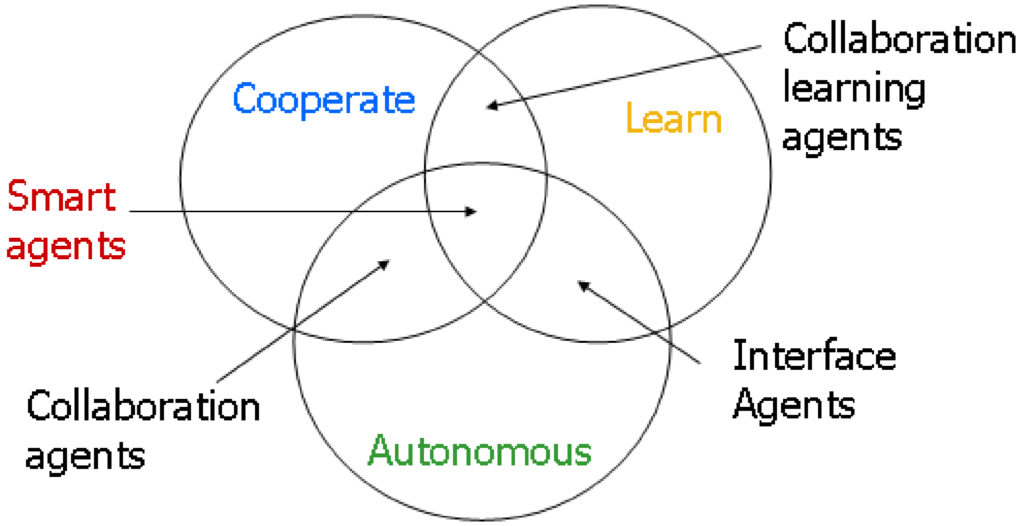


Figure 20 - Nwana's Category of Software Agents

Our “agents”, or objects, are able to cooperate and can act autonomously. If we develop a computational entity as controller of our system, the agent would be a collaboration agent. We could take a more difficult step to add learning possibilities. But this would require adding goals (we elaborate this idea in section 6.3.3) or a way to find out if something is a good or bad result.

Weiss suggests an abstract architecture for intelligent agents. This includes characterizing the agents' environment using S = {s1, s2, … } being environmental states. Then we assume the effectoric ability of an agent to be represented by a set A = { a1, a2, … } of actions. Then, abstractly an agent can be viewed as a function:

which maps the sequences of environment states to actions. Our system fits within this architecture. The environment can only change using actions or events, assuming for now we trigger events by other actions or the simulator "agent" (for example, using the clock). This architecture to describe an agent is an efficient but mathematical approach and is not suitable for domain experts.

Weiss also suggests adding a perception level for agents. This is something our GSDL simulator does not yet implement but might be a compulsory update if we were to program intelligent agents into the simulator. For now the perception each agent is the entire environment, while in reality each agent has its own perception. A quick implementation would be to allow agents only to precept objects that share the same location for example. For more elaborate functionality one should look at SDL described in section 2.2.2 which concentrates more on supporting an MAS then our language.

Another notation used to make MAS visually understandable is using Agent UML Notation (AML) (Huhns, 2004), stemmed from UML with its main purpose to support software development with more visual tools.

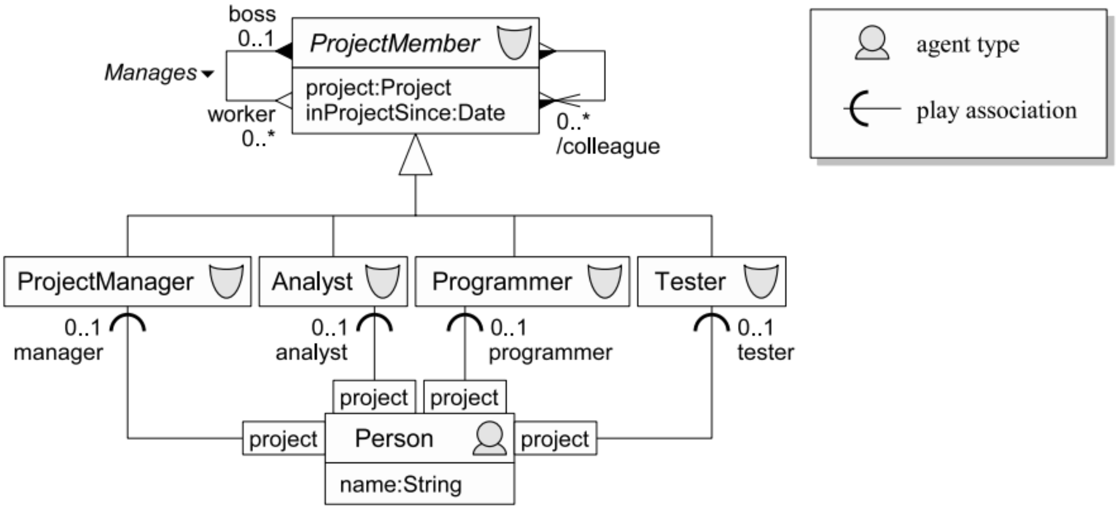


Figure 21 - AML notation example (Cervenka, Trencansky, & Calisti, 2006)

AML is created to support software programmers so it is too hard for domain experts to use. Possibly a simplified version would be a valuable added feature to help de domain expert understand the situation better then plain texts.

AML can aid the programmer by making the problem more visual. An interesting study would be if it is possible for our system to produce these kinds of models from our scenario definition automatically.

### 3.6.4 Compared with Labeled Transition Systems

We have mentioned LTS, or Labeled Transition Systems (Magee & Kramer, 2006). Our GSDL simulator has a few similarities so we tried to use the existing LTS knowledge to create a visual representation of our state machine. We came up with the following code:

```
const Max = 5
range L = 0..Max

AGENT = AGENT[0],
AGENT[location:L] = ( vehicle[location].enter -> DRIVER[location]
                                        | move_to[loc:L] -> AGENT[loc]
                                        ),
DRIVER[location:L] = ( vehicle[location].exit -> AGENT[location]
        | drive_to[loc:L] -> DRIVER[loc]
        ).

VEHICLE = VEHICLE[1],
VEHICLE[location:L] = ( vehicle[location].enter -> DRIVING[location]),
DRIVING[location:L] = ( vehicle[location].exit -> VEHICLE[location]
        | drive_to[loc:L] -> DRIVING[loc]
        ).
```

We found that, although these are also parallel processes that can enable or disable one another, it is not possible to create this code from a GSDL scenario. The code above resulted in the following state space diagram:

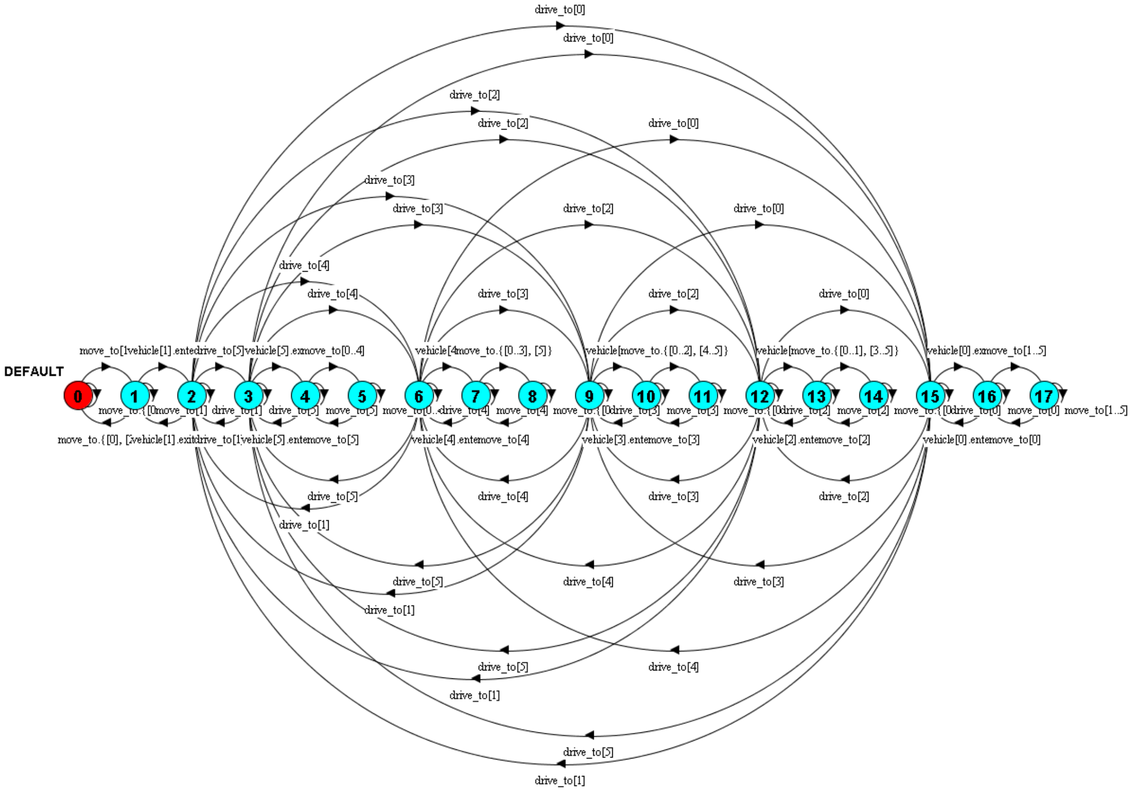

drive_to[0]
drive_to[0]
drive_to[2]
drive_to[2]
drive_to[0]
drive_to[3]
drive_to[3]
drive_to[2]
drive_to[0]
drive_to[4]
drive_to[4]
drive_to[3]
drive_to[2]
drive_to[0]
move_to[1vehicle[1].entedrive_to[5]vehicle[5].exmove_to[0..4]
vehicle[4move_to.{[0..3], [5]}
vehicle[move_to.{[0..2], [4..5]}
vehicle[move_to.{[0..1], [3..5]}
vehicle[0].exmove_to[1..5]
DEFAULT
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
move_to.{[0move_to[1] drive_to[1] drive_to[5] move_to[5] move_to[0..drive_to[4] move_to[4] move_to.{[0drive_to[3] move_to[3] move_to.{[0drive_to[2] move_to[2] move_to.{[0drive_to[0] move_to[0] move_to[1..5]
move_to.{[0], [2vehicle[1].exitdrive_to[1vehicle[5].entemove_to[5]
vehicle[4].entemove_to[4]
vehicle[3].entemove_to[3]
vehicle[2].entemove_to[2]
vehicle[0].entemove_to[0]
drive_to[5]
drive_to[4]
drive_to[3]
drive_to[2]
drive_to[1]
drive_to[5]
drive_to[4]
drive_to[3]
drive_to[1]
drive_to[5]
drive_to[4]
drive_to[1]
drive_to[5]
drive_to[1]

Figure 22 – A generated LTS state space diagram

Technically our system could create a visual LTS version of any scenario, like the one showed above. We just have to let the GSDL simulator perform each possible step. But we found hierarchical state machines more useful for our purposes. We discuss this in 0.

## 3.7 Conclusion Scenario Description Language

We used a simple scenario to show how we can use a scenario description language that uses the Java-syntax. It allows describing objects, relations, events and actions. We combined this with features of planning languages to allow testing GSDL code using a GSDL simulator. Using these base building blocks we claim that it is possible to describe any scenario in an efficient way. The detail of description can be varied when required.

The option to test the scenario before its implementation allows description and feedback of a scenario as required by **R1**. The restrictions on the scenario and interpretation of its elements make the description less ambiguous then a plain written description. We support interpreting variables by using them in the simulation; leaving less room for misinterpretations (thus answering **R2**). Separating domain and scenario allows the programmer to put some restrictions to the domain expert. But **R3** is not satisfied. We still allow the domain expert to add changes to the domain description. The advantage for the programmer is he can spot these changes easily and can act as a clear point of discussion for future meetings.

Because we also allow commentary, it is possible to add hints to the programmer or domain expert about implementation within the script if needed.

The main drawback of GSDL is that it is probably too difficult for the domain expert to use. So we need to provide a simple interface for the domain expert. The description language itself does not meet **R4** except for the extensive use. The solution to solve **R4** is the Scenario Description Tool we describe in section 4.

The object oriented approach using inheritance creates a modular and reusable description supporting **R5.** Once we create a solid base for a specific simulator, adding new scenarios will be a significant more efficient job because we can reuse large parts of the domain for multiple scenarios.

The practical environment described in section 3.2 shows back and forward communication as required by **R6**. The GSDL simulator allows some acknowledgments both ways. We aid communication by the possibility of the programmer to add extra tests and show the domain expert if the results are correct before implementing them in the simulator.

GSDL can be used as a base to solve knowledge gap communication problems. We show how this can be achieved in the following sections.

# 4 Scenario Description Tool

## 4.1 Introduction

To help the domain experts to generate GSDL code we provide them with a user interface to enter the values we need to create valid GSDL code. We developed GSDL in such a way that it is usable in a simulator, but also in an editor tool as we will show in this section.

Currently GSDL only adds limits to the syntax of the description. There are no checks to see if the code is valid. (For example, there is no check if the used variables are declared). Entering invalid code with correct syntax will result in a Java compile error on using the output files in the GSDL simulator. For a programmer these errors are easy to fix, but we want to restrain the domain expert from adding invalid code. A description tool can make sure all the code that is entered is valid by adding restrictions to the values that can be entered.

The interface can collect some more information for the domain expert to support describing the scenario. Although the programmer should make the first steps. The scenario description tool becomes more powerful when the domain definition becomes bigger.

Because of the time constraint of this project we did not implement the scenario description tool. In this section we show the features GSDL offers to ease creating such a tool and what advantages this offers for the domain expert.

## 4.2 Adding information to the domain definition

We first have to define the domain definition. A tool can provide a few checks to allow correct output.

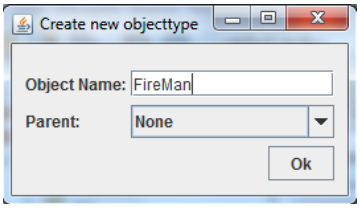
As described in section 3.3 we need the following values:

**Create a new type of Object:**

- objectname (Name of the object type)
- parentname (Optional, name of the parent of this object type).

The objectname needs to be unique, the tool can check for this by not allowing saving an existing name. The parentname needs to be an existing object; we can check this as well.

The next screen will show all the default variables (predicates, functions and messages) of the parent object. The user can choose to add or edit variables for this new object.



Object Name: FireMan

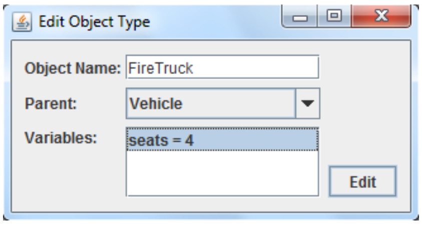
Parent: None

Figure 23 - Object Type Creation GUI

**Adding default variables to an Object:**

- name (String)
- type (Predicate, Message, Function)
- value.

The name of the variable needs to be unique for each object, no duplicates may exist. (See section 3.3). Each type of variable can only have one type of value. A predicate needs a true or false value, a message a string and a function a numeric value. The tool will be able to enforce the correct value type.



Object Name: FireTruck

Parent: Vehicle

Variables: seats = 4

Figure 24 - Object Type Editing GUI

**Create an event or action:**
The user can choose from the list of objects what objects we need for this action (prerequisites). For actions the user will need to select an initiator. It is possible to add more than one object of the same type. Each object will receive a variable name, by default this will be the name of the object type and a number but the user can choose to change this.

**Enabled function:**
Then we can edit the enabling function. We can only use objects added above to check variables from, we can offer to add new objects if the user tries to add invalid variables. There is always the simulator variable available to get simulator-wide variables like the time. The three types of object variables should deliver a true/false condition. We can do this as followed:

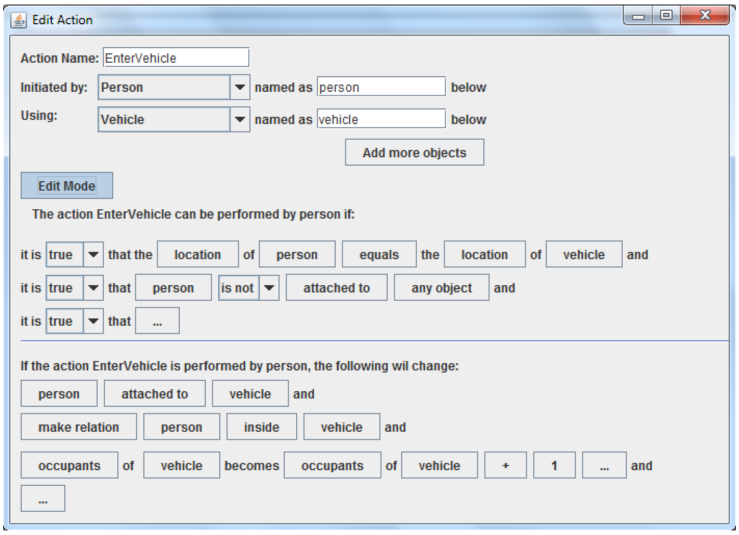
*Predicates:* are true or false by themselves. They can be negated using ! (NOT) operators.
*Relations:* are true if a relation exists.
*Functions:* can be numerically compared using =, <, >, <=, => operators. Values can also be operated by +, -, /,* operators.
*Messages:* can be compared using “equals” or “contains”.
*Attachments:* will result true if an object is attached (to a specific other object).

To aid the domain expert in defining complex formula's for predicates or functions the following addition to the tool can be useful. We can describe the formula in a readable sentence where the domain expert can add or change specific values. For example:

If it is true the speed of vehicle v1 is smaller than the speed of vehicle v2 + 10 … then this action can be performed by person p1 …

We can click on the underlined texts to change them or add more values. This way the domain expert can define more complex formulas. The example above would translate into

```
if (v1.speed < v2.speed + 10) return true;
```



Action Name: EnterVehicle

Initiated by: Person named as person below

Using: Vehicle named as vehicle below

The action EnterVehicle can be performed by person if:

it is true that the location of person equals the location of vehicle and

it is true that person is not attached to any object and

it is true that ...

If the action EnterVehicle is performed by person, the following wil change:

person attached to vehicle and

make relation person inside vehicle and

occupants of vehicle becomes occupants of vehicle + 1 ... and

...

Figure 25 - Action Editing GUI

**Perform function:**

Then we can edit the perform function. First we need to select an object to add, remove or change a variable.

The tool can aid the domain expert by adding extra (logical) security checks here. For example, if the domain expert is to use the attach option, the tool can check if the action is disabled when the object is already attached to another object. In our examples we could even automatically check if the location of both objects is the same before allowing an attachment between those two.

The combined interfaces above will allow the domain expert to create a domain definition. The chances are that the domain expert will not use this often since the programmer most likely already added most of the needed elements into the domain definition.

The domain expert will probably use more time creating the specific scenario and just using the domain definition. This would mean just adding instances of objects to the environment.

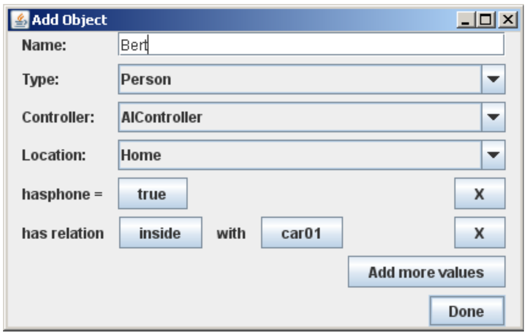
## 4.3 Scenario Description

The scenario description itself is the same as a perform function for actions and events. Important is the instantiation of new objects.

**Create an Object:**
The user enters the following compulsory information:
Name(String) – The name of the object
Type(Object) – The type of object, selected from a list of possible objects or the possibility to add a new type of object that would send the user to the "Create new type of Object" screen.
Controller (Controller) – The controller, defaults to AI-controlled.
Location (Location) – The location this object will start at the beginning of the simulation.

If the type of object is changes so will all the default fields of the object that we can fill in. We can add new values.

```
Person a = new Person("a"); // mandatory field, the unique name of the object
// optional fields can just be added when needed
a.setController(AIController);
a.setPredicate("hasphone", true);
```



Name: Bert
Type: Person
Controller: AIController
Location: Home
hasphone = true
has relation inside with car01

Entering the location would be different for 2D or 3D environments. Here we would present the user with a map or 3D world and let the user click to add the new location to the map. Also see section 6.3.

## 4.4 Usability

Using a Graphical User Interface allows the domain expert to work with GSDL. A tool like this offers some extra features to improve usability.

The tool can aid solving the scalability problems for scenarios. Currently if a scenario becomes bigger and more objects are available, the code will become obscure. It will become more difficult to preserve overview and much time can be lost searching for the right objects.
The tool can help here by grouping objects: for example, to allow mass editing large amounts of objects. The tool can also support searching for objects or actions.

In our examples we assumed a location is identified by a name only. For most simulators a location will exists within a 2D or 3D environment. Using a GUI like

Figure 6 - Valve Hammer Editor; adding objects to a scenario with these types of locations becomes possible. Also one could define certain areas. We can meet the conditions of actions if an object is present in a specific area. Or the initial position of an object can be defined as “anywhere within this area”.

## 4.5 Conclusion Scenario Description Tool

A Graphical User Interface (GUI) to produce GSDL code is necessary to allow the domain expert to work with GSDL. This tool will have to meet the following requirements:

- create a new type of object
- adding default variables to object types
- create an event or action
- adding objects to the scenario
- adding events to the scenario
- check for validity of the entered values
- produce valid GSDL code.

The following optional requirements will ease creating scenarios:

- allow grouping and mass editing multiple objects
- allow searching for objects or actions
- allow 2D or 3D locations and areas.

We presented suggestions and ideas about the implementation.

# 5 Evaluation

This section evaluates our findings. We will take the problems we found from section 1.2 and discuss how our solution contributed to solving those problems. The next part will address some issues that still needs coping with or are introduced by our solution.

## 5.1 Problem solving

A script that describes a scenario can aid the process of scenario generation. We reflect on the problems and show how we deal with them using our script.

**P0- It takes too much time and effort for a domain expert and programmer to obtain a shared understanding about implementing a scenario.**

**P1- Difference in jargon**.
We tackle the differences in jargon because the script allows a close to unambiguous description of the scenario because the domain expert tested whether this description is able to produce the expected results. Although variable names might still be confusing, we can derive the function of that variable from the script. This will result in less communication problems between both parties.

The scenario description tool for the domain expert will aid to speed up the process because the domain expert will not need to understand Java or GSDL.

**P2- Missing "obvious" information**.
The script can provide feedback to the domain expert through the GSDL simulator. In each state of the GSDL simulation the script can point out why a specific event can or cannot take place. This should prevent the domain expert from making these common mistakes, at least for the most obvious and desired course of events. We could reduce this problem even more if we add state-space exploration to the GSDL simulator (described in section 6.3.3). This would allow the domain expert to define invalid or error-states which can be discovered by the simulator.

**P3- Undefined restrictions.**
Our GSDL solution does not preserve strict control over restrictions of the simulator. This is because we do not want to confine the domain expert too much. The programmer can supply a domain definition that runs within the simulators boundaries. This way we can steer the domain expert to use objects within limits. By separating the domain and scenario we make it possible for the programmer to spot differences in important areas quickly by comparing his own working version of the domain with the new version.

**P4- Practical point of view.**
We cannot solve this entire problem. The GSDL simulator does allow a practical approach due the possibility of a test run. For unknown values to the domain

expert the option is open to leave a remark for the programmer to solve this problem.

**P5- Allow forbidden events to happen**.
Testing the scenario will prevent most forbidden events to happen; there might be some left when the domain expert is not able to explore all the states of the simulation. If we can define known pitfalls using the validity tests and state-space exploration (described in section 6.3.3) we can even check for these events automatically.

**P6- Abundance of information.**
We reduced the information needed to create a scenario as much as possible. The GSDL description is a compact set of all the information needed for a simulation. By reusing the domain definition and the choice to inherit objects we decrease the total amount of information that we need to add new information to a scenario. This will also avoid introducing new errors.

Using GSDL as proposed in our dissertation will speed up the development process due several reasons. The first reason is there is less need for discussion appointments. The domain expert will be able to tests and solve problems himself before asking the programmer. There are fewer contact moments and both parties can work more efficiently without each other.

Second reason is there is less time needed to program the simulator. Evaluation of the simulator is now mostly done in the GSDL simulator.

Third reason is the information delivered by the domain expert has a better quality; it is less error prone and it describes complete scenarios. When the programmer starts programming the simulator, the information available is more accurate than without GSDL.

## 5.2 Issues

We discovered some issues with our solution that we would like to address here and add suggestions on how to engage these issues.

**Untested**.
Our solution has no test results. We demonstrated the effect for small examples. We claim that our solution aids to solve our problem, backing this with theories, but whether this is true and to what degree it solves our problem is untested. We have some suggestions for tests in section 0. It is important to test if adding an extra tool saves more time that it costs to learn handling it.

**Scalability**.
We showed how we can simulate a simple scenario. For larger scenarios our GSDL might become obscure. In section 4.4 we suggest some possible solutions to this problem, but the general concern stays that large and complicated scenarios might become too difficult for the domain expert to understand.

Technically scalability might become a problem as well. From our definition we produce all possible actions when we load the GSDL simulator. For larger numbers of objects this will exponentially grow in numbers and runtime calculations. These are technical issues that can are solvable to a certain point.

**Concurrency.**
We used a discrete time simulation for our example. We also assumed that only one event or action may occur at any one timeslot. This is useful for simple scenario testing, but real life is continues and allows parallel processes to work concurrently. The goal of simulations within the emergency response domain is to train for real life.
It is possible to use GSDL for concurrent simulations, but we need to adjust it. We discuss some suggestions in section 3.6; for example, by adding durational actions and locking or synchronization mechanisms.

**Completeness.**
We claim that using GSDL it is possible to simulate any scenario. It is difficult to test whether that is true. We created an extensive language that supports a broad scale of choices for our scenario. Areas where our definition lacks support are time and space. For example, movement: we can think of a workaround, just like durational actions, by adding a "moving" predicate and increment the location by the value of its speed using events. But again this could prove to be too complicated for the domain expert. We can extend the GSDL language to support these kinds of essential properties of scenarios. We make some suggestions in section 6.3.2.

# 6 Conclusion and Future research

This section describes the conclusions of our research in section 6.1. Section 6.2 will present the answers to the research questions from section 1.4). We present ideas and improvements for future research in section 6.3.

## 6.1 Conclusion

We discovered that with scenario description for disaster scenarios too much time is lost to the communication process between the domain expert and the programmer. There is need for intense contact for both parties to understand each other. Misunderstandings result in faulty implementations and time-consuming reprogramming. This is a generic problem that occurs also in other domains.

The root to our solution to this problem is GSDL (Generic Scenario Description Language): a simple scripting language supported by a GSDL simulator and script writing tool. We showed the parsing process of a simple scenario and how this can be used for feedback in the GSDL simulator.

The language, based on object oriented and planning languages, is a compact form of information needed by a simulator to perform a simulation. The programmer can start by defining the domain definition. This is a piece of GSDL code that defines what objects might persist in the simulator and what actions these objects can perform. The programmer sends this to the domain expert, who, using the visual scenario description tool to load the domain definition, can start adding objects or instantiating them in the scenario. This scenario description tool can generate the required GSDL code for the domain expert. The code is automatically parsed to executable Java code for the included GSDL simulator. The GSDL simulator will show the domain expert what objects are present and what actions can be performed in the current state.

This feedback is decreases the communication process between the programmer and the domain expert. Figure 26 below shows the gain of time and number of meetings in a sequence diagram.

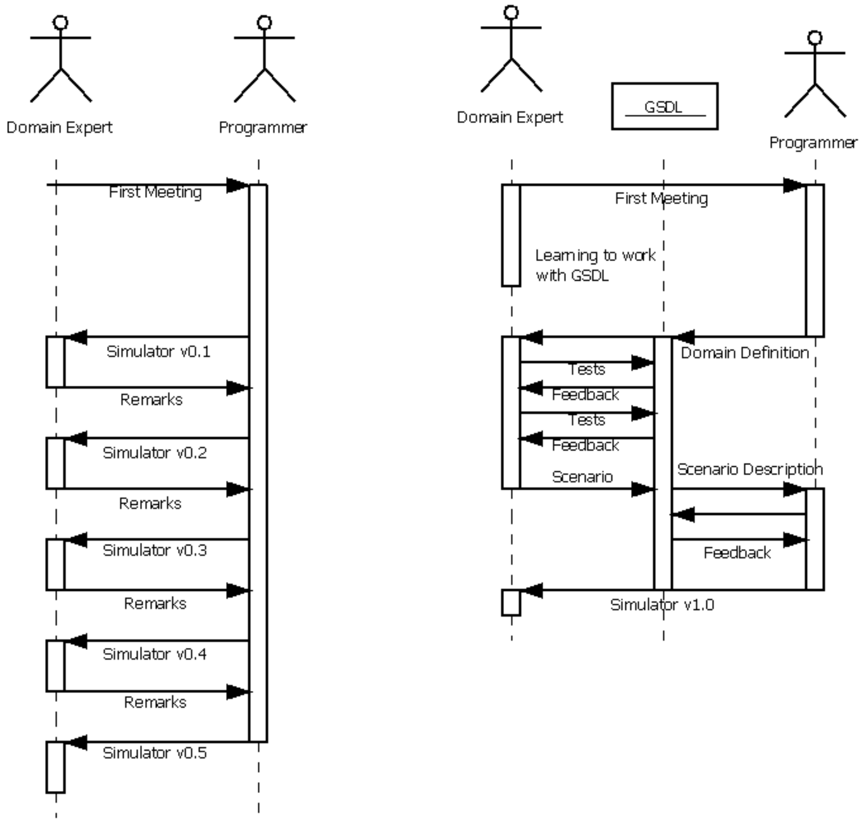


Figure 26 – Sequence Diagram GSDL use

Needing less time and fewer contact it is possible to make bigger development steps which results in an improvement of our previous situation.

It is to be tested if this theory works as we presume it will. We describe more on this in section 0. There are a few pitfalls that might lessen the success of GSDL we discussed these in section 5.2. It is important to find out if a domain expert is able to work efficiently with GSDL; this can be tested if the GUI (described in section 4) is functional.

The main advantages of using GSDL are:

- there is less need for appointments with the domain expert and programmer
- there is less time needed to program the simulator
- the quality of the information delivered by the domain expert is better.

## 6.2 Answers to the research questions

This section presents the answers found to our research questions:

**Q1- What problems define the knowledge gap problem with scenario generation for a computer simulation?**
The following problems define the knowledge gap problem; we explain them in section 1.2 and evaluate them in section 5.1.

**P0- It takes too much time and effort for a domain expert and programmer to get a shared understanding about implementing a scenario.**
**P1- Difference in jargon**.
**P2- Missing "obvious" information**.
**P3- Undefined restrictions.**
**P4- Practical point of view.**
**P5- Allow forbidden events to happen**.
**P6- Abundance of information.**

**Q2- What existing tools could offer a solution to any of these problems?**
We researched several existing tools in section 2. None of them could offer a decisive solution. We did found the following features that we used for our solution:

- allow the programmer to alter the domain expert's software program to enforce adding valid entries (2.1.3), supports reducing **P3** problems.
- allow the domain expert to add textual information to clarify things if needed (2.1.4), helps solving **P2** and **P5** problems.
- use a well-known object oriented script language to define the scenario (2.2.2); this aids solutions for **P6** problems.
- allow description of behavior (2.2.3), this helps us to solve **P2**, **P3** and **P5** problems.
- separate the domain definition from the scenarios (2.3.2), this helps reducing **P2** and **P6** problems.
- add actions with preconditions like planning languages to be able to define an extensive scenario without describing each state individually (2.3.2), this helps to use the GSDL simulator (which supports solutions to **P1**, **P2** and **P5**).This can also help solving **P5** with state space exploration (section 6.3.3).
- allow the programmer to define the boundary of the simulator (2.3.3) which helps solving **P3**.

The common problem with comparable techniques is that they are too complicated for nontechnical users. More information how these tools relate to our problems is described in section 2.

**Q3- Could (a combination of) these tools offer a solution to the general knowledge gap problem? How do these tools cope with the individual problems?**

Yes, we combined the methods to create GSDL which tackles most of the individual problems and logically also the general knowledge gap problem. We described the conclusion about coping with the individual problems in section 5.

We developed the GSDL toolset using the following requirements:

**R1- describe the entire scenario and support feedback;**
**R2- provide an unambiguous description of the desired scenario;**
**R3- allow a programmer to extend or restrict the template of a scenario;**
**R4- allow intuitive and extensive usability for the domain expert;**
**R5- be modular and reuse previously or predefined information;**
**R6- allow confirmation to both sides to confirm a similar understanding.**

Our solution exists of 3 tools, the GSDL language (described in section 3.4), that is build conform **R1**, **R2**, **R3** and **R5**; acts as a base for the communication. The GSDL simulator (described in section 3.5) uses this language to provide feedback to both the domain expert and the programmer to make sure there is a similar understanding about the scenario (supporting **R1**, **R2** and **R6**). Finally we suggest the creation of an intuitive scenario description tool for the domain expert to support **R4** (described in section 4).

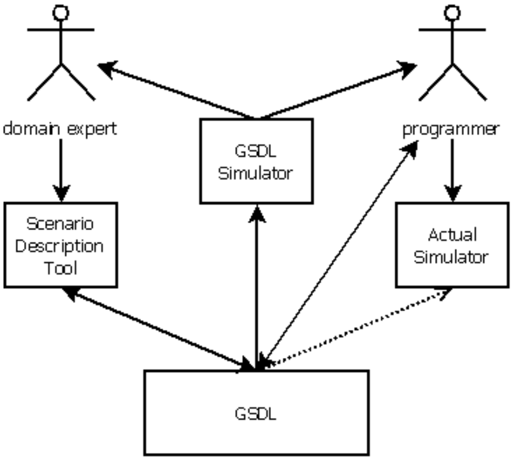


Figure 27 - GSDL Tools overview

The following table relates which requirements of our GSDL toolset aid solving which problems.

| | P1 | P2 | P3 | P4 | P5 | P6 |
|---|---|---|---|---|---|---|
| **R1** | | X | | | X | |
| **R2** | X | | | | X | |
| **R3** | | | X | | | |
| **R4** | X | | | X | | |
| **R5** | | | | | | X |
| **R6** | X | | X | | X | |

**Table 5 – Requirement versus Problem Table**

The main advantages of using GSDL are:

- there is less need for appointments with the domain expert and programmer
- there is less time needed to program the simulator
- the quality of the information delivered by the domain expert is better.

**Q4- How can we test effectiveness of the tools?**
Unfortunately we did not have time to perform tests. We will discuss how we could test our solution in our future research section 0.

## 6.3 Future research

This section elaborates a bit more on how to approach the steps we were not able to take for this dissertation. How to test if our solution is successful is described in section 0. Possible additions to the GSDL language are suggested in section 6.3.2. Extra features to our GSDL Simulator can be added using state space exploration which is described in section 6.3.3. How to use GSDL with other tools is described in section 6.3.4.
A feature we did implement in our demonstrated version is visualization using HSMs; we describe this in section 6.3.5.

### 6.3.1 Usability Tests

To test GSDL for emergency simulators we need programmers and domain experts. We can use GSDL for Generic Scenario Description. It is possible to use experts from other domains. The emergency response domain is fairly small.

**General testing**

We claim that using GSDL will speed up the scenario creation process. To test this, multiple test groups should perform the same task of creating and implementing the same scenarios. Some groups will use GSDL and others will just communicate intensively. Elements that need testing are:

- time used communicating
- time used understanding GSDL
- time used developing scenario (domain expert)
- time used implementing scenario (programmer)
- completeness of the implementation
- number of misunderstandings
- experience of the users.

It could be that for simple scenarios the saving of time is the minimum. The assumption is that with more complicated or a larger number of scenarios the saving of time will play a significant role.

We cannot test the added features of state exploration and transition diagrams because these are all added values for the domain expert.

**Testing the language**

It is important to test if the language is complete enough to describe each scenario and if it is compatible with the large set of simulators. We can ask the programmers if it is easy to understand the script and if it is possible to add a scenario described in this script to their simulator.

We can ask the domain experts using the GUI if they can use the object/action-based structure to create the scenario they have in mind.

**Testing the GUI**

The GUI needs to meet the requirements of general usability tests and international standards (ISO 9241). Testing of the GUI can be done with both domain experts and programmers, but the tool is mainly created to be used by domain experts.

It is advisable to do a requirement research before creating the GUI. The domain experts should provide input before testing. This act of requirements engineering can be used as a test for the GSDL language as well, this will point out if GSDL can be used efficiently.

### 6.3.2 GSDL v2

As discussed in our evaluation section (5.2) we can extend the current version of GSDL to support some more essential features for real life scenario description. These include:

- durational actions
- continues time-space
- location operators
- dimensional operators
- movement operators.

These features will allow using GSDL for currently existing simulators like ADMS or other simulators discussed in our previous research (Jeeninga, 2010). We can add most of these to the grammar file to support these actions in the GSDL script. For example, object sizes:

```
dimensionDeclaration ::=
'dimension' 'x' '=' NUMBER ';'
'dimension' 'y' '=' NUMBER ';'
'dimension' 'z' '=' NUMBER ';';
```

Snippet 9 - EBNF suggestion for dimensions in GDSL v2

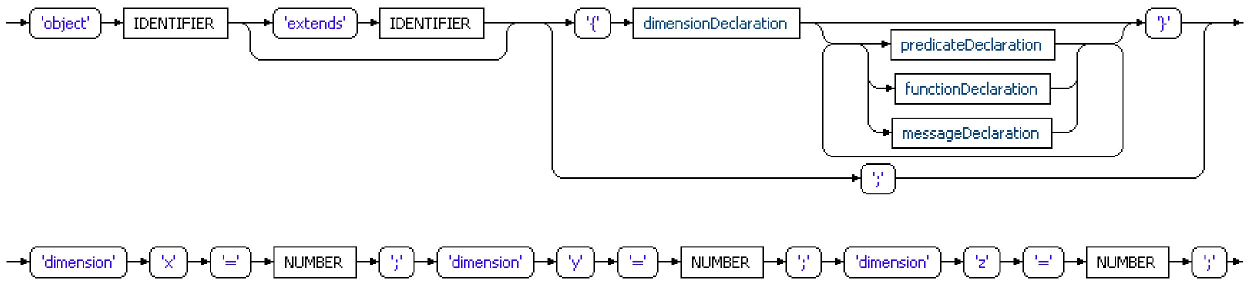
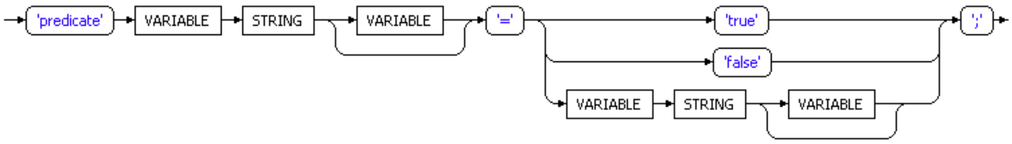


Figure 28 - Railroad notation for dimensions in GSDL v2

Looking back at section 2.1.3, using dimensions we could use GSDL to produce an FGD file for the Valve Hammer Editor™, allowing scenario generation in a 3D environment. See also section 6.3.4 for more ideas on this matter.

We could also optimize the GSDL simulator we created to support faster calculations for scenarios. We could only update the actions that changed during the last adaptation and cache the rest. Options to save states and scenarios could help speed up the GSDL simulator use. This would solve some scalability issues.

We also discovered we can describe the relations between objects as predicates. Due time restriction this was not implemented in the demonstrated version of GSDL, but we can simplify the language if we allow adding names of objects to the name of a predicate. For example:



This would allow entering the following code:

```
predicate person "inside" vehicle = true;
```

This would translate to the following Java code:

```
person.setPredicate("inside" + vehicle.getName(), true);
```

And we can test it with:

```
person.getPredicate("inside" + vehicle.getName());
```

This would simplify the GSDL language because there is even less syntax used. The GUI could keep the relations interface only the translation to GSDL is different. One should keep in mind that this could restrict some possibilities as well; if you want to check if a person has a relation "inside" with any object for example. This might be solved using messages and its "contains" function but this can produce some extra problems that needs solving.

### 6.3.3 State-space exploration

We can see the current discrete time simulator as a finite state system. The simulator can calculate a starting state from the given scenario and create new states from there. The current implementation has a simple hashing algorithm for each state. It is possible to detect whether a state is already visited, this way it is possible to explore each possible state that we can visit beginning at our starting state.

Now we can define a goal. For example, a certain value to be true, an action to be enabled, or a combination of these goals to be reached. Using our explored state-space we can select the path to the closest state where these predicates are true, providing an optimal solution for our goal.

Oppositely it is possible for the domain expert to add restrictions to the scenario. For example, when it is not possible in the reality for two values to be true at the same time, we can warn the domain expert if there are states reachable where this is the case. The scenario needs extending to avoid these error-states.

Using continues time-space will hinder state space exploration because we are not dealing with a finite state system anymore. It might be possible to be able to use the advantages of a finite state model. We would need to remove some elements from distinguishing states (time for example) and cut the continued time in discrete pieces (using the enabled functions that deal with time)

### 6.3.4 Compatibility

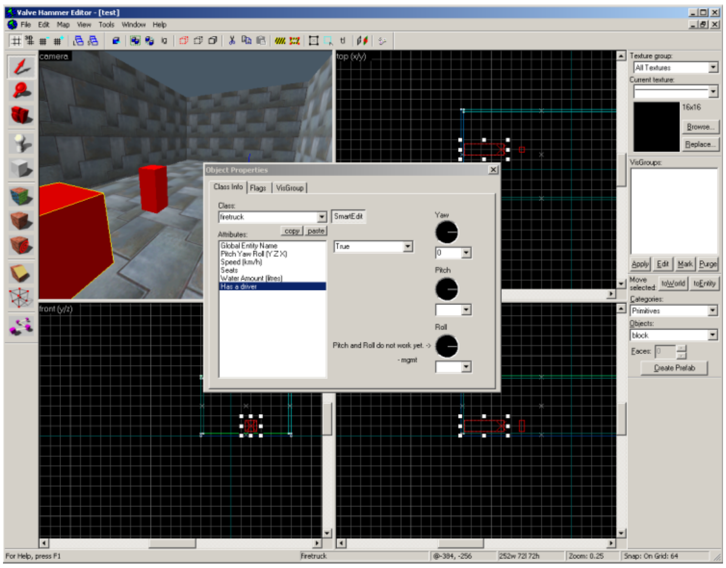
To explain the use for our language we can use an existing 3D-environment editing tool (for example, the Valve Hammer Editor). We use the domain definition from our description language to produce an FGD-file for the editor. This process is a one-on-one change of the domain definition.

There are some minor features that we can adjust to make it work more flawless with the editor. For example, we can tell our conversion tool how we defined the size of the objects. If this is not defined by the domain definition we will need to use default values or we can ask for user input on this matter.
Here is how we convert our fire truck from the script to FGD format:

| | |
|---|---|
| **Base** | |
| `public BaseObject(String name) {`<br>`        this.name = name;`<br>`}` | `@BaseClass = Global`<br>`[`<br>`        globalname(string) : "Global Entity Name"`<br>`]` |
| **Domain** | |
| `public class Vehicle extends BaseObject {`<br><br>`        public Vehicle(String name) {`<br>`                super(name);`<br>`        }`<br><br>`}` | `@BaseClass base(Global) = Vehicle` |
| `public class FireTruck extends Vehicle {`<br><br>`        public FireTruck(String name) {`<br>`                super(name);`<br>`        }`<br><br>`}` | `@PointClass base(Vehicle) = firetruck : "Firetruck"`<br>`[]` |

For example, if we add variables like water capacity it would change like this:

| | |
|---|---|
| `public class FireTruck extends Vehicle {`<br><br>`   public FireTruck(String name, Float capacity) {`<br>`        super(name);`<br>`        this.setFunction("capacity", capacity);`<br>`   }`<br>`}` | `@PointClass base(Vehicle) = firetruck : "Firetruck"`<br>`[`<br>`        capacity(string) : "capacity" : ""`<br>`]` |



Class: firetruck

Attributes:
Global Entity Name
Pitch Yaw Roll (Y Z X)
Speed (km/h)
Seats
Water Amount (litres)
Has a driver

True

Yaw: 0

Pitch and Roll do not work yet. ->
- mgmt

Figure 29 - the use of our description language in VHE

Using the editor we can place objects in the scenario and set their initial values like the location, but also the predicates and functions that are available because of the domain definition. We can compile this information into a map-file that we can convert to a scenario written in our own script again.

As discussed in section 5.2, one point needs to be redesigned for compatibility with multiple other simulators: the simulator specific information like time, location and object size. For example, to support checks if objects are present in a certain three-dimensional area.

Compatibility with existing description languages is important to look at. SDL (section 2.2.2) has several common features with GSDL. An interesting research would be if we can use our simple language to automatically gather complicated scenarios in SDL. If this is true then we could directly use GSDL to aid scenario programming for users of simulators that use SDL. We can do the same for MSDL; we can describe the predefined list in a domain description and use this with GSDL.

### 6.3.5 Visualization using state transition diagrams

A carried out feature in our demonstration version is the Hierarchical State Machine (HSM) visualization. This can aid to supply overview in difficult and large-scale simulations. We can model the internal state of any object using the HSM tool within the GSDL simulator to help see internal problems. Internal states are common practice in simulators, as you can see by the Unreal Script™ in section 2.1.3.

For example, if a truck can contain two passengers, we can model its internal state as follows:

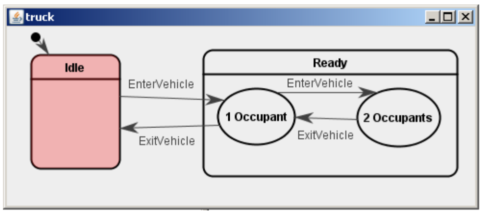


Figure 30 - truck HSM model within our GSDL simulator

While running the scenario, the state graph will change if we invoke an action EnterVehicle using the truck as parameter. If we invoke EnterVehicle twice in a row without an ExitVehicle action the truck is full and will produce an error if the scenario allows another EnterVehicle action.

We manually modeled this example HSM state. It is possible to derive states from the simulation, but the graphical placement of the states might be difficult. It would result in a diagram like Figure 22 – A generated LTS state space diagram (page 57). Also added information like the state-names needs to be produced and might seem obscure.

It can be good practice for the simulation programmer to use HSM models for complicated objects. We can use this to validate the scenario.

We discussed AML models earlier in section 3.6.3. It would be interesting to see if we can use more models like this for visualization of our GSDL simulator.

# 7 References

Cervenka, R., Trencansky, I., & Calisti, M. (2006). *Modeling Social Aspects of Multi-Agent Systems: The AML Approach.* Bratislava, Slovakia: Springer.

Dahl, O.-J., & Nygaard, K. (1965). *Basic concepts of SIMULA: an ALGOL based simulation language.*

Hendrix, M., & Brederode, F. (2009, 06 10). E-semble interview. Paris.

Huhns, M. N. (2004). *Agent UML Notation for Multiagent System Design.* IEEE Computer Society.

Hyacinth, S. N. (1996). *Software Agents: An overview.* Cambridge University Press.

Ishida, T. (2005, November). Q: A Scenario Description Language for Interactive Agents. *Computer* .

Jeeninga, J. (2010). *Investigation of Scenario Simulation.* Enschede.

Kobossen, B. (2010, February 15). NIFV ADMS Interview. Amesfoort.

Kramer, E., Mensink, M., & Osinga, A. (2009, May 26). Berhloo. *TriMM interview*.

Magee, J., & Kramer, J. (2006). *Concurrency State Models and Java Programs.* Wiley.

Nau, D. e. (2003). SHOP2: An HTN Planning System. *Journal of Artificial Intelligence Research 20* , pp. 379-404.

Nilsson, R. F. (1971). STRIPS: a new approach to the application of theorem proving to problem solving. *Artificial Intelligence* , pp. 2:189-208.

Nusman, D. (2009, March 23). RE-lion ANWB driving simulator. *RE-lion interview*.

Olaya, A. G. (2007, September). Seminario PLG: Overview of PDDL.

Parr, T. (2007). *The Definitive ANTLR Reference: Building Domain-Specific Languages.*

Porskamp, P., & Groot, T. d. (2009, 06 24). Txchange interview. Enschede.

Quartel, S. (2009). *Projectplan voor ISETICfP HOP221.*

Shortliffe, E. (2006). *Biomedical Informatics Computer Applications in Health Care and Biomedicine.*

SISO. (sd). *Simulation Interoperability Standarads Organization*. Opgehaald van http://www.sisostds.org/

Sweeney, T. (2010). *UnrealScript Language Reference .* Epic Games.

Weiss, G. (1999). *Multiagent Systems.* Cambridge, Massachusetts: The MIT Press.

Widding, E. (2009, 11 17). Fire department interview.

Willemsen. (2000). *Behaviour amd Scenario Modeling for Real-time Virtual Environments.* Iowa.

World Wide Web Consortium. (sd). *W3C XML Schema*. Opgeroepen op 2010, van W3C: http://www.w3.org/XML/Schema

# 8 Appendices

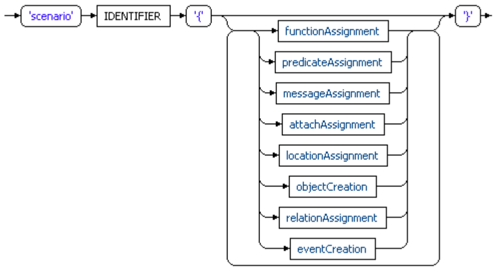
## 8.1 EBNF Description of the Scenario Description Language

```
actionDeclaration ::= 'action' IDENTIFIER '{' initiatorDeclaration utilizationDeclaration
(enabledFunction performFunction | performFunction enabledFunction) '}' ;

eventDeclaration ::= 'event' IDENTIFIER '{' utilizationDeclaration
(enabledFunction performFunction | performFunction enabledFunction) '}' ;

initiatorDeclaration ::= 'initiator' IDENTIFIER VARIABLE ';' ;

utilizationDeclaration ::= 'utilize' IDENTIFIER VARIABLE ';' ;

enabledFunction ::= 'enabled' '{' ('true' ';' |
(relationExpression | messageExpression | attachedExpression | otherExpression | locationExpression)+ ) '}' ;

performFunction ::= 'perform' '{' (functionAssignment | predicateAssignment | messageAssignment | attachAssignment |

locationAssignment | objectCreation | relationAssignment)* '}' ;

relationExpression ::= NOT VARIABLE 'hasrelation' STRING ('with'? VARIABLE)? ';' ;

messageExpression ::= NOT VARIABLE STRING ('equals' | 'contains') (VARIABLE STRING | STRING) ';' ;

otherExpression ::= expression RELATIONALOPERATOR expression ';' ;

attachedExpression ::= NOT VARIABLE 'is' 'attached' ';' ;

locationExpression ::= NOT VARIABLE 'location' 'equals' VARIABLE 'location'? ';' ;

predicateAssignment ::= 'predicate' VARIABLE STRING '=' ('true' | 'false' | VARIABLE STRING) ';' ;

relationAssignment ::= 'relation' ('make' | 'remove') VARIABLE STRING VARIABLE ';' ;

functionAssignment ::= 'function' VARIABLE STRING '=' expression ';' ;

messageAssignment ::= 'message' VARIABLE STRING '=' (VARIABLE STRING | STRING) ';' ;

attachAssignment ::= ('attach' VARIABLE 'to'? VARIABLE ';' | 'detach' VARIABLE ';') ;

locationAssignment ::= 'location' VARIABLE '=' VARIABLE ';' ;

objectCreation ::= IDENTIFIER VARIABLE ('=' STRING)? ';' ;

eventCreation ::= 'create'? 'event' IDENTIFIER '(' VARIABLE (',' VARIABLE)* ')' ';' ;

expression ::= (partialExpression | (VARIABLE STRING | NUMBER | 'time') OPERATOR expression |
(VARIABLE STRING | NUMBER | 'true' | 'false' | 'time')) ;

partialExpression ::= '(' expression ')' ;

predicateDeclaration ::= 'predicate' STRING '=' ('true' | 'false') ';' ;

functionDeclaration ::= 'function' STRING '=' NUMBER ';' ;

messageDeclaration ::= 'message' STRING '=' STRING ';' ;

IDENTIFIER ::= UPPERCASE_CHARACTER (LOWERCASE_CHARACTER | UPPERCASE_CHARACTER | DIGIT | '_')* ;

VARIABLE ::= LOWERCASE_CHARACTER (LOWERCASE_CHARACTER | UPPERCASE_CHARACTER | DIGIT | '_')* ;

UPPERCASE_CHARACTER ::= 'A..Z' ;

LOWERCASE_CHARACTER ::= 'a..z' ;

RELATIONALOPERATOR ::= ('==' | '!=' | '<=' | '>=' | '<' | '>') ;

OPERATOR ::= ('+' | '-' | '/' | '*' | '%' | '^') ;

NOT ::= '!' ;

STRING ::= '"' ~('\\' | '"' | '\r' | '\n') '"' ;

NUMBER ::= ('-'? '1..9' DIGIT* ('.' DIGIT+ )? | '0') ;

DIGIT ::= '0..9' ;

WS ::= (' ' | '\t' | '\n' | '\r' | '\f')+ ;

COMMENT ::= '//' .* ('\n' | '\r') ;
```

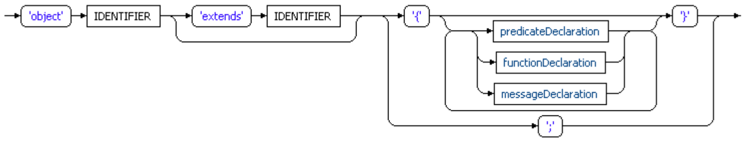
http://en.wikipedia.org/wiki/Extended_Backus%E2%80%93Naur_Form

## 8.2 GSDL Railroad view

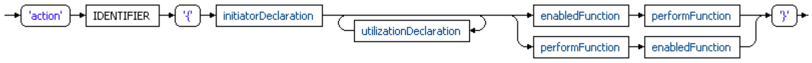
scenarioDeclaration



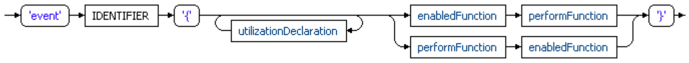
objectDeclaration



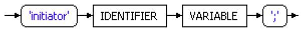
actionDeclaration



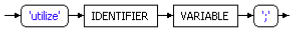
eventDeclaration



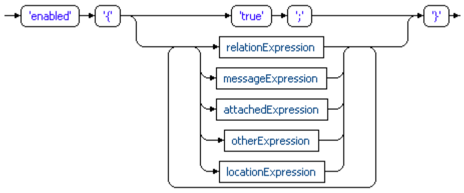
initiatorDeclaration



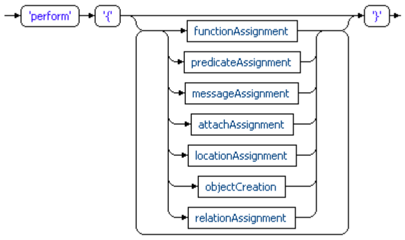
utilizationDeclaration



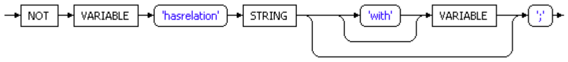
enabledFunction



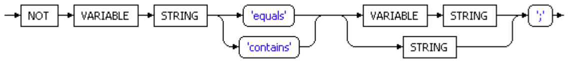
performFunction



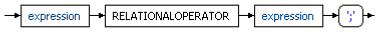
relationExpression



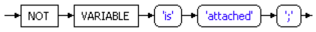
messageExpression



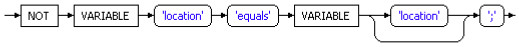
otherExpression



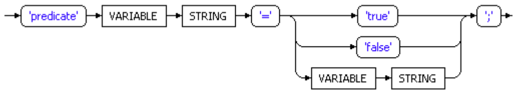
attachedExpression



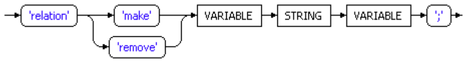
locationExpression



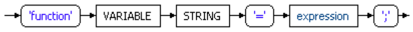
predicateAssignment



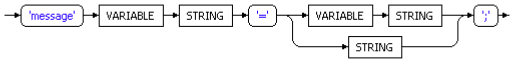
relationAssignment



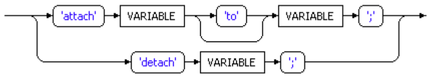
functionAssignment



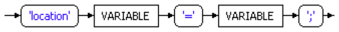
messageAssignment



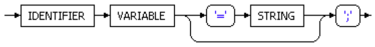
attachAssignment



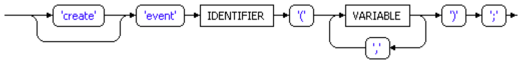
locationAssignment



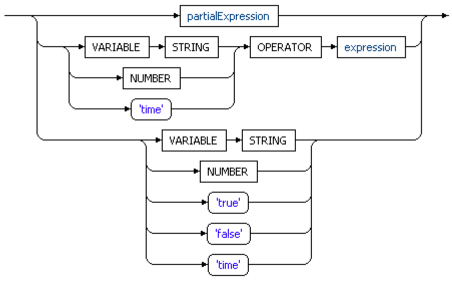
objectCreation



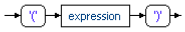
eventCreation



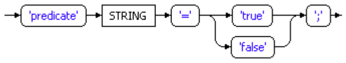
expression



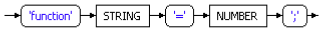
partialExpression



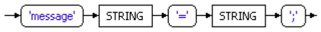
predicateDeclaration



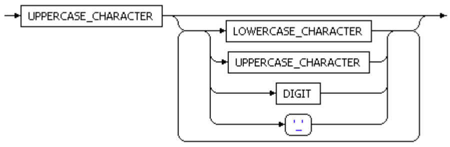
functionDeclaration



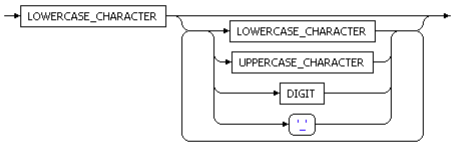
messageDeclaration



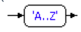
IDENTIFIER



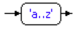
VARIABLE



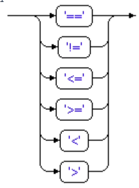
UPPERCASE_CHARACTER



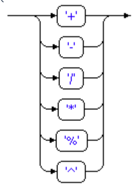
LOWERCASE_CHARACTER



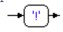
RELATIONALOPERATOR



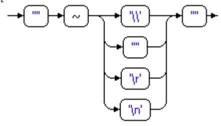
OPERATOR



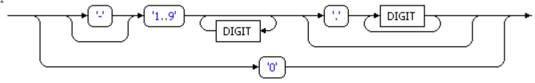
NOT



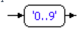
STRING



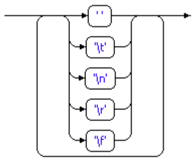
NUMBER



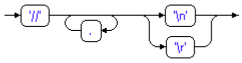
DIGIT



WS



COMMENT



## 8.3 Template API

GSDL Template API

This API describes what variables will be passed to which files in the StringTemplate of the GSDL parser so one can create a custom GSDL convertor. For more information on the GSDL language please see the GSDL EBNF notation.

Usage: change the files from the GSDL templates directory to support your custom output.

Required files in the template directory:

| | |
|---|---|
| `object.st` | Defines the output of an object declaration |
| `action.st` | Defines the output of an action declaration |
| `event.st` | Defines the output of an event declaration |
| `expression.st` | Defines the output of an expression |
| `predicatedeclaration.st` | Defines the output for a predicate declaration |
| `functiondeclaration.st` | Defines the output for a function declaration |
| `messagedeclaration.st` | Defines the output for a message declaration |

The input you receive from the parser for each of the template files is described below.

All values are passed without quote-signs. If you want to escape or strip quotes you added for example, in the expression.st template file, use the format="escapeQuotes" or format="stripQuotes" functions of the supplied BasicFormatRenderer.

### Object.st

| Variable | Type | Description |
|---|---|---|
| `$objectname$` | `String` | The name for this object. |
| `$parentname$` | `String` | Parent name for this object, this value is unset if there is no parent object. |
| `$predicates$` | `List` | A list of predicates using the predicatedeclaration template. |
| `$functions$` | `List` | A list of functions using the functiondeclaration template. |
| `$messages$` | `List` | A list of messages using the messagedeclaration template. |

### Action.st

| Variable | Type | Description |
|---|---|---|
| `$actionname$` | `String` | The name for this action. |
| `$initiator$` | `String` | The variable name followed by a space character and then followed by the type of object that is the initiator object for this action. |
| `$utilize$` | `List` | A list of variable names and their types of |

| | | |
|---|---|---|
| | | the other objects that are needed for this action. |
| `$variables$` | `List` | A list of all variables, only the variable names in order of appearance. |
| `$enabled$` | `List(Expression)` | A list of expressions that enable this action. See the expression template. |
| `$perform$` | `List(Assignment)` | A list of assignments that should be performed when this action is fired. See assignment below. |

**Assignment**

An assignment can be of various types. Therefore each assignment has one unique variable that is set to "true" to be able to check the type of assignment.

| Variable | Type | Description |
|---|---|---|
| `$predicate$` | `String` | This variable is set to "true" if this is a predicate assignment |
| `$message$` | `String` | This variable is set to "true" if this is a message assignment |
| `$function$` | `String` | This variable is set to "true" if this is a function assignment |
| `$relation$` | `String` | This variable is set to "true" if this is a relation assignment |

**Predicate Assignment**

| Variable | Type | Description |
|---|---|---|
| `$predicate$` | `String` | This variable is set to "true" if this is a predicate assignment |
| `$var1$` | `String` | The variable name of the object to set this predicate. |
| `$predicatename$` | `String` | The name of the predicate |
| `$bool$` | `String` | This variable is only set when the predicate is set by a direct boolean. This variable is "true" or "false" depending on the setting. |
| `$var2$` | `String` | The variable name of the object where we derive our predicate from. This is only set in combination with $predicatename2$ if $bool$ is not set. |
| `$predicatename2$` | `String` | The name of the predicate to derive the value from. |

**Message Assignment**

| Variable | Type | Description |
|---|---|---|
| `$message$` | `String` | This variable is set to "true" if this is a message assignment |
| `$var1$` | `String` | The variable name of the object to set this message. |
| `$message1$` | `String` | The name of the message |
| `$var2$` | `String` | This variable is only set when the message depends on another variable, otherwise the $message2$ should be the message. |
| `$message2$` | `String` | The variable name of the object where we derive our message from. Or if $var2$ is not set, this is our message. |

**Function Assignment**

| Variable | Type | Description |
|---|---|---|
| `$function$` | `String` | This variable is set to "true" if this is a function assignment |
| `$var1$` | `String` | The variable name of the object to set this function. |
| `$function1$` | `String` | The name of the function |
| `$expression$` | `String` | The expression that defines the value of this function. See expression for more info. |

**Relation Assignment**

| Variable | Type | Description |
|---|---|---|
| `$relation$` | `String` | This variable is set to "true" if this is a relation assignment |
| `$var1$` | `String` | The variable name of the object to set this relation. |
| `$relation1$` | `String` | The name of the relation |
| `$make$` | `String` | This variable is set to "true" if a relation is to be made, this variable is not set if a relation is to be removed. |
| `$var2$` | `String` | The variable name of the object to make the relation with. |

## Event.st

| Variable | Type | Description |
|---|---|---|
| `$eventname$` | `String` | The name of this event. |
| `$variables$` | `List` | A list of all the variable names used in this event. |
| `$utilize$` | `List` | A list of all variables used, followed by a space followed by the objecttype of the variable. |
| `$enabled$` | `List(Expression)` | A list of expressions that enable this action. See the expression template. |
| `$perform$` | `List(Assignment)` | A list of assignments that should be performed when this event is fired. See assignment from Action.st above. |

## Expression.st

There are 3 types of expressions. Relation expressions which have the $relation$ variable set, message expressions which have the $message$ variable set and other expressions.

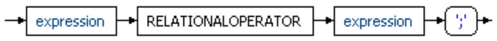
### Relation Expression

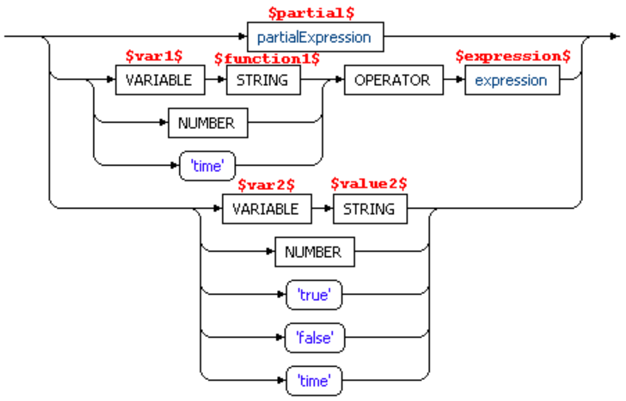
| Variable | Type | Description |
|---|---|---|
| `$var1$` | `String` | The variable to check this relation on. |
| `$relation$` | `String` | The name of the relation. |
| `$with$` | `String` | If this variable is set we want to check whether the object defined in $var1$ has a relation with the object defined in $with$, else we want to know if the object has any relation called $relation$. |
| `$not$` | `String` | This value is set if we want to negate the result. |

### Message Expression

| Variable | Type | Description |
|---|---|---|
| `$var1$` | `String` | The variable to get this message from. |
| `$message1$` | `String` | The name of the message. |
| `$var2$` | `String` | If we want to compare the message to another object his message we define this in $var2$ |
| `$message2$` | `String` | This is the name of the message in case $var2$ is set, otherwise just use the value of this variable. |
| `$contains$` | `String` | This value is set if we want to check if $message1$ contains $message2$, check for equal values otherwise. |
| `$not$` | `String` | This value is set if we want to negate the result. |

### Other Expression





The other expressions are a little complicated. A lot of values can be set and in that case others are not. The railroad-view above helps visualize which values are used. For example, if $var1$ and $function1$ are set, $expression$ is also set.

The difference from var1 and var2 is that for var1 it is known that this is a function-expression since there is an operator involved. $var2$ could be a message, boolean or a function.

| Variable | Type | Description |
|---|---|---|
| `$partial$` | `String` | A partial expression, recursively using this template and preceded by the (-sign and ended with a )-sign. |
| `$var1$` | `String` | A variable name to get a function from, this value is unset if this expression does not use a function value. |
| `$function1$` | `String` | The function name to get the value from using $var1$. This value is unset if $var1$ is unset. |
| `$number$` | `String` | This is a plain numeric value to compare with the expression after the operator, this value is set if $var1$ and $partial$ is unset. |
| `$operator$` | `String` | The operator for this expression, can be any of +-/*%^ |
| `$expression$` | `String` | The expression after the operator, recursively using this template. |
| `$var2$` | `String` | The variable to get the value from, this can be a boolean or function variable. |
| `$value2$` | `String` | The name of the value to receive from $var2$, this value is not set if $var2$ is not set. |
| `$bool$` | `Integer` | A number representing "true" or "false" for boolean comparison. This value becomes "1" or "0" respectively. |
| `$time$` | `String` | This will stand for the current time, a numeric value that can be compared to functions. |

Example:

```
$if(partial)$$partial$
$elseif(var1)$$var1$.getFunction("$function1$")
$elseif(var2)$$var2$.getValue("$value2$")
$elseif(number)$$number$f
$elseif(bool)$$bool$f
$endif$
$if(operator)$ $operator$ $expression$
$endif$
```

## predicatedeclaration.st

| Variable | Type | Description |
|---|---|---|
| `$predicate$` | `String` | Name of the predicate to declare |
| `$value$` | `String` | The value to set for this predicate (true or false) |

## functiondeclaration.st

| Variable | Type | Description |
|---|---|---|
| `$function$` | `String` | Name of the function to declare |
| `$value$` | `String` | The value to set for this predicate (numeric value) |

## messagedeclaration.st

| Variable | Type | Description |
|---|---|---|
| `$message$` | `String` | Name of the message to declare |
| `$value$` | `String` | The value to set for this message (String) |

## 8.4 Lexer and Parser ANTRL Grammar

```
grammar GSDL;

options {
  language = Java;
}

/* PARSER RULES */

@header {
  import org.antlr.stringtemplate.*;
  import org.antlr.stringtemplate.language.*;
  import java.util.HashMap;
}

@members {
StringTemplateGroup templates =
    new StringTemplateGroup("GSDLGroup", System.getProperty("user.dir") +"/src/templates");
}

parseAll returns [HashMap result]
@init {
  result = new HashMap<String, String>();
}
: (
  object=objectDeclaration { result.put("domain" + System.getProperty("file.separator") + "objects" +
System.getProperty("file.separator") + object.getAttribute("objectname"), object.toString());}
  |action=actionDeclaration { result.put("domain" + System.getProperty("file.separator") + "actions" +
System.getProperty("file.separator") + action.getAttribute("actionname"), action.toString());}
  |event=eventDeclaration { result.put("domain" + System.getProperty("file.separator") + "events" +
System.getProperty("file.separator") + event.getAttribute("eventname"), event.toString());}
  |scenario=scenarioDeclaration { result.put("scenario" + System.getProperty("file.separator") +
scenario.getAttribute("scenarioname"), scenario.toString());}
)*;

scenarioDeclaration returns [StringTemplate scenarioTemplate]
@init {
  scenarioTemplate = templates.getInstanceOf("scenario");
  ArrayList<HashMap> assignments = new ArrayList<HashMap>();
}
: 'scenario' name=IDENTIFIER '{'
  (
  (a=functionAssignment|a=predicateAssignment|a=messageAssignment|a=attachAssignment
  |a=locationAssignment|a=objectCreation|a=relationAssignment|a=eventCreation){ assignments.add( a ); }
  )*
'}'
{
  scenarioTemplate.setAttribute("scenarioname", $name.text);
  scenarioTemplate.setAttribute("assignments", assignments);
};

objectDeclaration returns [StringTemplate objectTemplate]
@init {
  //Use the object.st template to create the code for this object
  objectTemplate = templates.getInstanceOf("object");
}
: 'object' objectname=IDENTIFIER ('extends' parentname=IDENTIFIER)?
(('{'
  ((p=predicateDeclaration|f=functionDeclaration|m=messageDeclaration)
  {
    if (p != null) objectTemplate.setAttribute("predicates", p);
    if (f != null) objectTemplate.setAttribute("functions", f);
    if (m != null) objectTemplate.setAttribute("messages", m);
    p = null;
    f = null;
    m = null;
  })*
'}')|';')
{
  objectTemplate.setAttribute("objectname", $objectname.text);
  objectTemplate.setAttribute("parentname", $parentname.text);
};

actionDeclaration returns [StringTemplate actionTemplate]
@init {
  //Use the action.st template to create the code for this object
  actionTemplate = templates.getInstanceOf("action");
  //Use our custom BasicFormatRenderer to allow the escapeQuote format.
  actionTemplate.registerRenderer(String.class, new BasicFormatRenderer());
}
: 'action' actionname=IDENTIFIER '{'
  i=initiatorDeclaration { actionTemplate.setAttribute("variables", i.get(1));
actionTemplate.setAttribute("initiator", i.get(0) + " " + i.get(1)); }
  (u=utilizationDeclaration { actionTemplate.setAttribute("variables", u.get(1));
actionTemplate.setAttribute("utilize", u.get(0) + " " + u.get(1)); })*
  (e=enabledFunction p=performFunction
  |p=performFunction e=enabledFunction) { actionTemplate.setAttribute("enabled", e);
actionTemplate.setAttribute("perform", p); }
  '}'
{
actionTemplate.setAttribute("actionname", $actionname.text);
};

eventDeclaration returns [StringTemplate eventTemplate]
@init {
  //Use the event.st template to create the code for this object
  eventTemplate = templates.getInstanceOf("event");
  //Use our custom BasicFormatRenderer to allow the escapeQuote format.
  eventTemplate.registerRenderer(String.class, new BasicFormatRenderer());
}
: 'event' eventname=IDENTIFIER '{'
  (u=utilizationDeclaration { eventTemplate.setAttribute("variables", u.get(1));
eventTemplate.setAttribute("utilize", u.get(0) + " " + u.get(1)); })*
  (e=enabledFunction p=performFunction
  |p=performFunction e=enabledFunction) { eventTemplate.setAttribute("enabled", e);
eventTemplate.setAttribute("perform", p); }
  '}'
{
eventTemplate.setAttribute("eventname", $eventname.text);
};

initiatorDeclaration returns [List<String> init]
// return a list of the two values we need. first one is the identifier and the second the variable
: 'initiator' i=IDENTIFIER v=VARIABLE { init = new ArrayList<String>(); init.add($i.text);
init.add($v.text); }';';
utilizationDeclaration returns [List<String> util]
: 'utilize' u=IDENTIFIER v=VARIABLE { util = new ArrayList<String>(); util.add($u.text); util.add($v.text);
}';';

enabledFunction returns [List expressions]
@init {
  expressions = new ArrayList<HashMap>();
}
: 'enabled' '{'
  ('true' ';'{ HashMap<String,String> h = new HashMap<String,String>(); h.put("expression", "true");
expressions.add( h ); }
  |( r=relationExpression { expressions.add( r ); }
  | m=messageExpression { expressions.add( m ); }
  | a=attachedExpression { expressions.add( a );}
  | o=otherExpression { expressions.add( o ); }
  | l=locationExpression { expressions.add( l );}
  )+)
  '}';

performFunction returns [List assignments]
@init {
  assignments = new ArrayList<HashMap>();
}
: 'perform' '{'
  (
  (a=functionAssignment|a=predicateAssignment|a=messageAssignment|a=attachAssignment
  |a=locationAssignment|a=objectCreation|a=relationAssignment){ assignments.add( a ); }
  )*
'}';

relationExpression returns [HashMap<String, String> relation]
@init {
  relation = new HashMap<String, String>();
}
: n=NOT? v=VARIABLE 'hasrelation' r=STRING ('with'? v2=VARIABLE)? ';'
{
  relation.put("relation", $r.text);
  relation.put("var1", $v.text);
  if ($v2 != null) {
    relation.put("with", $v2.text);
  }
  if ($n != null) {
    relation.put("not", "!");
  }
};

messageExpression returns [HashMap<String, String> message]
@init {
  message = new HashMap<String, String>();
}
: n=NOT? v1=VARIABLE s1=STRING op=('equals'|'contains') (v2=VARIABLE s2=STRING|s2=STRING) ';'
{
  message.put("message", "true");
  message.put("var1", $v1.text);
  message.put("message1", $s1.text);
  message.put("message2", $s2.text);
  if(op.getText().equals("contains")) {
    message.put("contains", "true");
  }
  if ($n != null) {
    message.put("not", "!");
  }
  if ($v2 != null) {
    message.put("var2", $v2.text);
  }
};


otherExpression returns [HashMap<String, String> expression]
@init {
expression = new HashMap<String,String>();
}
: (e1=expression op=RELATIONALOPERATOR e2=expression ';') { expression.put("expression", e1 + " " +
$op.text + " " + e2); };

attachedExpression returns [HashMap<String, String> attached]
@init {
  attached = new HashMap<String, String>();
}
: n=NOT? v1=VARIABLE 'is' 'attached' ';'
{
  attached.put("attached", $v1.text);
  if ($n != null) {
    attached.put("not", "!");
  }
};


locationExpression returns [HashMap<String, String> expression]
@init {
  expression = new HashMap<String, String>();
}
: n=NOT? v1=VARIABLE 'location' 'equals' v2=VARIABLE 'location'? ';'
{
  expression.put("location", "true");
  if ($n != null) {
    expression.put("not", "!");
  }
  expression.put("var1", $v1.text);
  expression.put("var2", $v2.text);
};

predicateAssignment returns [HashMap<String, String> assignment]
@init {
  assignment = new HashMap<String, String>();
}
: ('predicate' v1=VARIABLE s1=STRING '=' (b='true'|b='false'|v2=VARIABLE s2=STRING) ';')
{
  assignment.put("predicate", "true");
  assignment.put("var1", $v1.text);
  assignment.put("predicatename", $s1.text);
  assignment.put("bool", $b.text); //b.equals("true") ? "1f" : "0f");
  assignment.put("var2", $v2.text);
  assignment.put("predicatename2", $s2.text);
};

relationAssignment returns [HashMap<String,String> assignment]
@init {
  assignment = new HashMap<String,String>();
}
: ('relation' action=('make'|'remove') v1=VARIABLE s1=STRING v2=VARIABLE ';')
{
  assignment.put("relation", "true");
  if($action.text.equals("make")) assignment.put("make", "true");
  assignment.put("var1", $v1.text);
  assignment.put("relation1", $s1.text);
  assignment.put("var2", $v2.text);
};

functionAssignment returns[HashMap<String,String> assignment]
@init {
  assignment = new HashMap<String, String>();
}
: ('function' v1=VARIABLE s1=STRING '=' e=expression ';')
{
  assignment.put("function", "true");
  assignment.put("var1", $v1.text);
  assignment.put("function1", $s1.text);
  assignment.put("expression", e);
};

messageAssignment returns [HashMap<String,String> assignment]
@init {
  assignment = new HashMap<String,String>();
}
```

```
: ('message' v1=VARIABLE s1=STRING '=' (v2=VARIABLE s2=STRING|s2=STRING) ';')
{
  assignment.put("message", "true");
  assignment.put("var1", $v1.text);
  assignment.put("message1", $s1.text);
  assignment.put("var2", $v2.text);
  assignment.put("message2", $s2.text);
};

attachAssignment returns [HashMap<String, String> assignment]
@init {
  assignment = new HashMap<String, String>();
}
: ('attach' v1=VARIABLE 'to'? v2=VARIABLE ';') {
  assignment.put("attach", "true");
  assignment.put ("var1", $v1.text);
  assignment.put ("var2", $v2.text);
   }
| ('detach' v1=VARIABLE ';') {
  assignment.put("detach", "true");
  assignment.put("var1", $v1.text);
   }
{

};

locationAssignment returns [HashMap<String, String> assignment]
@init {
  assignment = new HashMap<String, String>();
}
: ('location' v1=VARIABLE '=' v2=VARIABLE ';')
{
  assignment.put("location", "true");
  assignment.put("var1", $v1.text);
  assignment.put("var2", $v2.text);
};

objectCreation returns [HashMap<String, String> assignment]
@init {
  assignment = new HashMap<String, String>();
 }
 : type=IDENTIFIER v1=VARIABLE ('=' name=STRING)? ';'
{
  assignment = new HashMap<String, String>();
  assignment.put("object", "true");
  assignment.put("type", $type.text);
  assignment.put("var1", $v1.text);
  assignment.put("name", $name.text);
};

eventCreation returns [HashMap<String, String> assignment]
@init {
  assignment = new HashMap<String, String>();
  StringTemplate eventCreateTemplate = templates.getInstanceOf("eventcreation");
}
: 'create'? 'event' name=IDENTIFIER '(' v=VARIABLE? (',' v=VARIABLE?)* ')' ';'
{
  eventCreateTemplate.setAttribute("name", $name.text);
  eventCreateTemplate.setAttribute("variables", $v.text);
  assignment.put("eventcreate", eventCreateTemplate.toString());
};

expression returns [String expression]
@init {
  StringTemplate expressionTemplate = templates.getInstanceOf("expression");
}
:
  (
  p=partialExpression { expressionTemplate.setAttribute("partial", p); }
  |(v=VARIABLE s=STRING { expressionTemplate.setAttribute("var1", $v.text);
expressionTemplate.setAttribute("function1", $s.text); }
    |n=NUMBER { expressionTemplate.setAttribute("number", $n.text); }
    |'time' { expressionTemplate.setAttribute("time", "true");})
    o=OPERATOR e=expression { expressionTemplate.setAttribute("operator", $o.text);
expressionTemplate.setAttribute("expression", e); }
  |(v=VARIABLE s=STRING {expressionTemplate.setAttribute("var2", $v.text);
expressionTemplate.setAttribute("value2", $s.text);}
    |n=NUMBER { expressionTemplate.setAttribute("number", $n.text);}
    |'true' { expressionTemplate.setAttribute("bool", "1");}
    |'false' { expressionTemplate.setAttribute("bool", "0");}
    |'time' { expressionTemplate.setAttribute("time", "true");})
  )
{
  expression = expressionTemplate.toString();
};

partialExpression returns [String expression]: '(' e=expression ')' { expression = "( " + e + " )"; };
```

```
predicateDeclaration returns [String result]
@init {
  StringTemplate predicateTemplate = templates.getInstanceOf("predicatedeclaration");
}
: 'predicate' p=STRING '=' v=('true'|'false') ';'
{
  predicateTemplate.setAttribute("predicate", $p.text);
  predicateTemplate.setAttribute("value", $v.text);
  result = predicateTemplate.toString();
};

functionDeclaration returns [String result]
@init {
  StringTemplate functionTemplate = templates.getInstanceOf("functiondeclaration");
}
: 'function' f=STRING '=' v=NUMBER ';'
{
  functionTemplate.setAttribute("function", $f.text);
  functionTemplate.setAttribute("value", $v.text);
  result = functionTemplate.toString();
};

messageDeclaration returns [String result]
@init {
  StringTemplate messageTemplate = templates.getInstanceOf("messagedeclaration");
}
: 'message' m=STRING '=' v=STRING ';'
{
  messageTemplate.setAttribute("message", $m.text);
  messageTemplate.setAttribute("value", $v.text);
  result = messageTemplate.toString();
};


/* LEXER RULES */
IDENTIFIER
    :   UPPERCASE_CHARACTER (LOWERCASE_CHARACTER | UPPERCASE_CHARACTER | DIGIT | '_')*
    ;

VARIABLE
    : LOWERCASE_CHARACTER (LOWERCASE_CHARACTER | UPPERCASE_CHARACTER | DIGIT | '_')*
    ;

fragment UPPERCASE_CHARACTER: 'A'..'Z';

fragment LOWERCASE_CHARACTER: 'a'..'z';

RELATIONALOPERATOR
    : '=='
    | '!='
    | '<='
    | '>='
    | '<'
    | '>'
    ;

OPERATOR
  : '+'
  | '-'
  | '/'
  | '*'
  | '%'
  | '^';

NOT : '!';

STRING : '"' ~( '\\' | '"' | '\r' | '\n' )+ '"' {setText(getText().substring(1, getText().length()-1));};
// Strip the "'s from the string

NUMBER: ('-'? '1'..'9' DIGIT* ('.' DIGIT+)?)|'0';

fragment DIGIT : '0'..'9';

WS : (' ' | '\t' | '\n' | '\r' | '\f')+ {$channel = HIDDEN;};
COMMENT : '//' .* ('\n'|'\r') {$channel = HIDDEN;};
```

## 8.5 Code snippets

### BaseAction

```
package base;


public abstract class BaseAction {

        public BaseAction() {
                Simulator.addAction(this);
        }
        public abstract boolean enabled();
        public abstract void perform();

        public abstract String enabledStatus();

       @Override
        public String toString() {
                String s =this.getClass().getName().substring(15) + " ";
                return s;
        }
}
```

### BaseEvent

```
package base;

public abstract class BaseEvent {

        public BaseEvent() {
                Simulator.addEvent(this);
        }

        public abstract boolean enabled();
        public abstract void perform();
       public abstract String enabledStatus();
}
```

### BaseObject

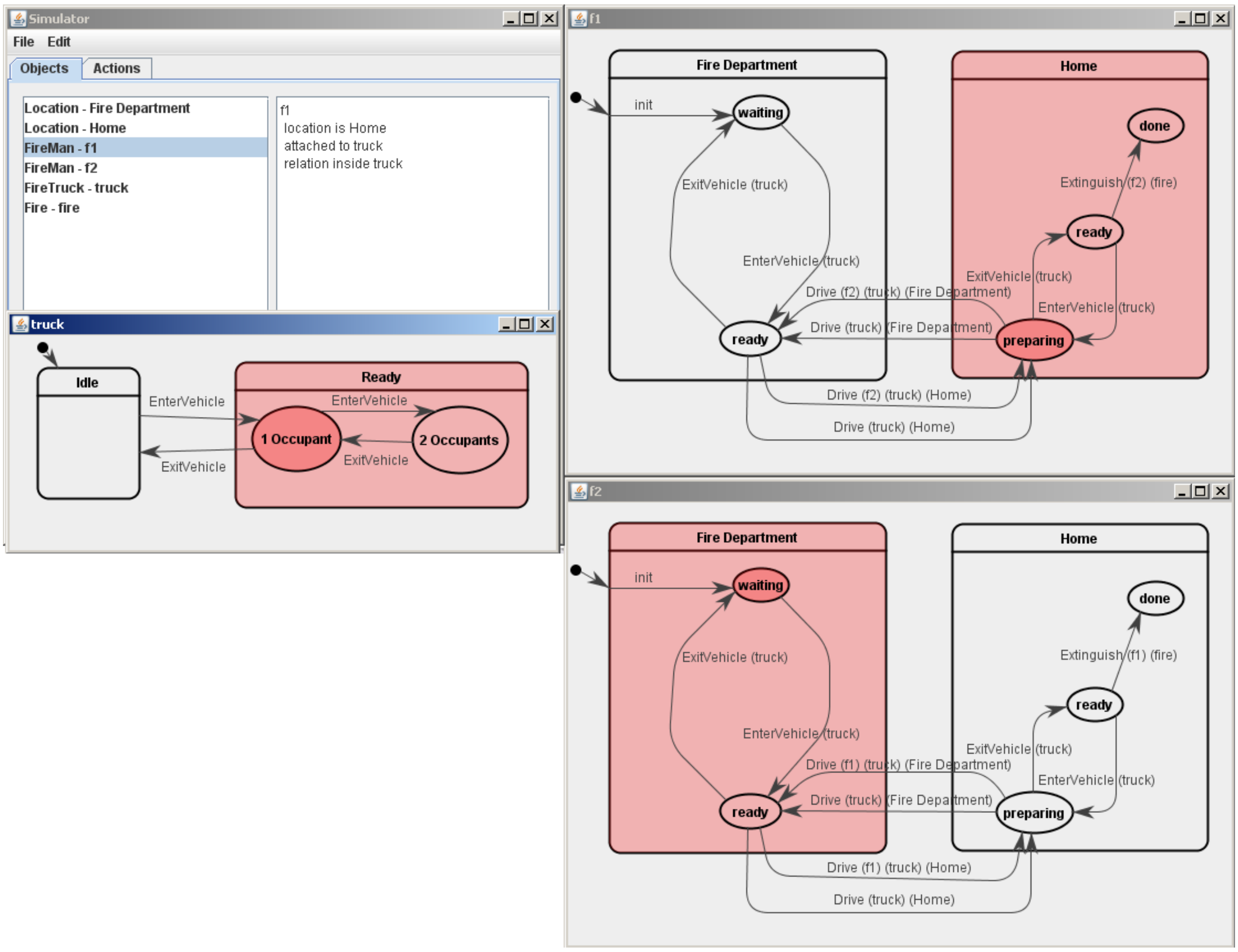
```
package base;

import domain.objects.Location;
import java.util.HashMap;
import java.util.Map;
import java.util.Vector;
import java.util.logging.Logger;

/**
 * @author  jeeninga
 */
public abstract class BaseObject {

    private static Logger log = Logger.getLogger("base");

    private final String name;

    private Location location = null;
    private BaseObject attached = null;

    private Controller controller = new BaseController();
    private HashMap<String, Boolean> predicates = new HashMap<String, Boolean>();
    private HashMap<String, Float> functions = new HashMap<String, Float>();
    private HashMap<String, Vector<BaseObject>> relations = new HashMap<String, Vector<BaseObject>>();
    private HashMap<String, String> messages = new HashMap<String, String>();
    private HashMap<String, VariableType> typeList = new HashMap<String, VariableType>();
    private Vector<String> actions = new Vector<String>();

    public BaseObject(String name) {
        this.name = name;
        this.init();
        Simulator.addObject(this);
    }

    /**
     * Method to initialize default values.
     */
    protected void init() {
    }

    ;

    /**
     * @return name
     */
    public final String getName() {
        return this.name;
    }

    /**
     * @param c
     */
    public final void setController(Controller c) {
        this.controller = c;
    }

    /**
     * @return
     */
    public final Controller getController() {
        return this.controller;
    }

    public final Location getLocation() {
        if (attached != null) {
            return attached.getLocation();
        }
        return this.location;
    }

    public final void setLocation(Location l) {
        if (attached != null) {
            log.warning("Could not change location of object " + name + "; object is attached to " + attached.getName());
            return;
        }
        this.location = l;
    }

    /**
     * Attach this object to another object, this means the location of this object will stay the same as the location of the
provided object until detached
     * @param o the object to attach this object to.
     */
    public final void attachToObject(BaseObject o) {
        if (attached != null) {
            log.warning("Could not attach object " + name + "; object is attached to " + attached.getName());
            return;
        }
        this.attached = o;
    }

    /**
     * Detach this object. The location will change to the last known location of the object this was attached to.
     */
    public final void detachFromObject() {
        if (attached == null) {
            log.warning("Trying to detach object " + name + "; object is not attached");
            return;
        }
        this.location = attached.location;
        attached = null;
    }

    /**
     * A boolean to check if this object is attached to any other object
     * @return
     */
    public final boolean isAttached() {
        if (attached == null) {
            return false;
        }
        return true;
    }

    public final boolean getPredicate(String name) {
        if(!predicates.containsKey(name)) {
            log.warning("Could not find predicate " + name);
        }
        return predicates.get(name);
    }

    public final void setPredicate(String name, Boolean value) {
        if (this.typeList.containsKey(name)) {
            if (!(this.typeList.get(name) == VariableType.PREDICATE)) {
                log.warning("Could not add predicate " + name + "; " + name + " already exists as type " +
this.typeList.get(name));
                return;
            }
        }
        else {
            this.typeList.put(name, VariableType.PREDICATE);
        }
        this.predicates.put(name, value);
        this.controller.notify(this);
    }

    public final HashMap<String, Boolean> getPredicates() {
        return this.predicates;
    }

    public final Float getFunction(String name) {
        if (functions.containsKey(name)) {
            return functions.get(name);
        }
        return null;
    }

    public final void setFunction(String name, Float value) {
        if (this.typeList.containsKey(name)) {
            if (!(this.typeList.get(name) == VariableType.FUNCTION)) {
                log.warning("Could not add function " + name + "; " + name + " already exists as type " +
this.typeList.get(name));
                return;
            }
        }
        else {
            this.typeList.put(name, VariableType.FUNCTION);
        }
        this.functions.put(name, value);
        this.controller.notify(this);
    }

    public final HashMap<String, Float> getFunctions() {
        return this.functions;
    }

    public final void setMessage(String name, String message) {
        if (this.typeList.containsKey(name)) {
            if (!(this.typeList.get(name) == VariableType.MESSAGE)) {
                log.warning("Could not add message " + name + "; " + name + " already exists as type " +
this.typeList.get(name));
                return;
            }
        }
        else {
            this.typeList.put(name, VariableType.MESSAGE);
        }
        this.messages.put(name, message);
        this.controller.notify(this);
    }

    public final String getMessage(String name) {
        if (messages.containsKey(name)) {
            return messages.get(name);
        }
        return null;
    }

    public final HashMap<String, String> getMessages() {
        return this.messages;
    }

    public final boolean hasRelationWith(String name, BaseObject object) {
        if (!this.relations.containsKey(name)) {
            return false;
        }
        for (BaseObject o : this.relations.get(name)) {
            if (object == null) {
                return true;
            }
            if (o.equals(object)) {
                return true;
            }
        }
        return false;
    }

    public final boolean hasRelation(String name) {
        return this.relations.containsKey(name);
    }

    public final void removeRelationWith(String name, BaseObject object) {
        if (!this.relations.containsKey(name)) {
            return;
        }
        if (object == null) {
          this.removeRelation(name);
        }
        else if (this.relations.get(name).contains(object)) {
            this.relations.get(name).remove(object);
            this.controller.notify(this);
        }
    }

    public final void removeRelation(String name) {
        if (!this.relations.containsKey(name)) {
            return;
        }
        this.relations.remove(name);
        this.controller.notify(this);
    }

    public final void setRelation(String name, BaseObject object) {
        if (this.typeList.containsKey(name)) {
            if (!(this.typeList.get(name) == VariableType.RELATION)) {
                log.warning("Could not add relation " + name + "; " + name + " already exists as type " +
this.typeList.get(name));
                return;
            }
        }
        else {
            this.typeList.put(name, VariableType.RELATION);
        }
        if (!this.relations.containsKey(name)) {
            Vector<BaseObject> v = new Vector<BaseObject>();
            v.addElement(object);
            this.relations.put(name, v);
        }
        else {
            this.relations.get(name).addElement(object);
        }
        this.controller.notify(this);
    }

    public final Float getValue(String name) {
        if (this.typeList.containsKey(name)) {
            if (this.typeList.get(name) == VariableType.PREDICATE) {
                if (this.predicates.get(name)) {
                    return 1f;
                }
                else {
                    return 0f;
                }
            }
            if (this.typeList.get(name) == VariableType.FUNCTION) {
                return this.functions.get(name);
            }
        }
        else {
            log.warning("getValue called on non existent value");
        }
        return 0f;
    }

    public final HashMap<String, Vector<BaseObject>> getRelations() {
        return this.relations;
    }

    public final boolean hasAction(String name) {
        return this.actions.contains(name);
    }

    public final void addAction(String name) {
        if (this.actions.contains(name)) {
            return;
        }
        this.actions.addElement(name);
        this.controller.notify(this);
    }

    public final void removeAction(String name) {
        if (!this.actions.contains(name)) {
            return;
        }
        this.actions.removeElement(name);
    }

    public final VariableType getVariableType(String name) {
        return this.typeList.get(name);
    }

    @Override
    public final String toString() {
        return this.getClass().getName().substring(15) + " - " + this.getName();
    }

    public final String getStringState() {
        String s = this.name + "\n";
        if (this.getLocation() != null) s += " location is " + this.getLocation().getName() + "\n";
        if (this.attached != null) s += " attached to " + this.attached.getName() + "\n";
        for (Map.Entry<String, Boolean> entry : this.predicates.entrySet()) {
            s += " predicate " + entry.getKey() + " is " + entry.getValue() + "\n";
        }
        for (Map.Entry<String, Float> entry : this.functions.entrySet()) {
            s += " function " + entry.getKey() + " is " + entry.getValue() + "\n";
        }
        for (Map.Entry<String, String> entry : this.messages.entrySet()) {
            s += " message " + entry.getKey() + " is " + entry.getValue() + "\n";
        }
        for (Map.Entry<String, Vector<BaseObject>> entry : this.relations.entrySet()) {
            for (BaseObject b : entry.getValue()) {
                s += " relation " + entry.getKey() + " " + b.getName() + "\n";
            }
        }
        return s;
    }

    /**
     * A very simple hashing method. Just put all the parameters in one large string
     * @return a string containing all parameters of this object.
     */
    private final String getHashString() {
        String s = this.name;
        if (this.getLocation() != null) s += this.getLocation().getName();
        if(this.attached != null) s += this.attached.getName();
        for (Map.Entry<String, Boolean> entry : this.predicates.entrySet()) {
            //ignore the false values since all entries are false by default
            if(entry.getValue()) s += entry.getKey() + entry.getValue();
        }
        for (Map.Entry<String, Float> entry : this.functions.entrySet()) {
            s += entry.getKey() + entry.getValue();
        }
        for (Map.Entry<String, String> entry : this.messages.entrySet()) {
            s += entry.getKey() + entry.getValue().hashCode();
        }
        for (Map.Entry<String, Vector<BaseObject>> entry : this.relations.entrySet()) {
            for (BaseObject b : entry.getValue()) {
                s += entry.getKey() + b.getName();
            }
        }
        return s;
    }

    @Override
    public final int hashCode() {
        return this.getHashString().hashCode();
    }
    @Override
    public final boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final BaseObject other = (BaseObject) obj;
        if ((this.name == null) ? (other.name != null) : !this.name.equals(other.name)) {
            return false;
        }
        return true;
    }
}
```

# GSDL Simulator

```java
package base;

import java.io.File;
import java.lang.reflect.Constructor;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Vector;
import java.util.logging.ConsoleHandler;
import java.util.logging.Level;
import java.util.logging.Logger;

import java.net.URL;
import java.util.HashSet;

import scenario.*;

//import scenario.Scenario1;
/**
 * @author    jeeninga
 */
public class Simulator {

    public static Logger log = Logger.getLogger("base");
    /**
           */
    private static int time = 0;
    private static int stateId = 0;
    /**
           */
    private static Vector<BaseObject> objects = new Vector<BaseObject>();
    private static Vector<BaseEvent> events = new Vector<BaseEvent>();
    /**
           */
    private static Vector<BaseAction> actions = new Vector<BaseAction>();
    private static HashMap<Integer, String> statespace = new HashMap<Integer, String>();

    public static void main(String[] args) {
        // Create a console handler
        ConsoleHandler handler = new ConsoleHandler();
        handler.setLevel(Level.ALL);
        log.addHandler(handler);
        log.setLevel(Level.ALL);

        Scenario1 s = new Scenario1();
        s.init();
        actions = new Vector<BaseAction>();
        try {
            for (Class<?> c : Simulator.getClasses("domain.actions")) {
                Simulator.checkAction(c);
            }
        }
        catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
        performEvents();
        java.awt.EventQueue.invokeLater(new Runnable() {

            public void run() {
                new Gui().setVisible(true);
            }
        });

    }

    /**
     * Make a discrete Simulation step. This means increasing the time and performing events
     */
    public static void step() {
        time++;
        performEvents();
    }

    private static void performEvents() {
        for (BaseEvent e : Simulator.events) {
            if (e.enabled()) {
                log.fine("Performing event " + e.toString());
                e.perform();
            }
        }
    }

    /**
     * Returns a string representation of all the object, actions and events with their parameters in the current state.
     * @return string representation of the current state.
     */
    public static String stringState() {
        String s = new String();
        for (BaseObject o : Simulator.objects) {
            s += o + "\n";
        }
        for (BaseEvent e : Simulator.events) {
            s += e + "\n";
        }
        for (BaseAction a : Simulator.actions) {
            s += a + (a.enabled() ? " = enabled" : " = disabled") + "\n";
        }
        return s;
    }
    /**
           * Returns the current time of the Simulation.
           * @return
           */
    public static int getTime() {
        return new Integer(time);
    }

    /**
     * Add an object to the Simulation.
     * @param o the object to add.
     */
    public static void addObject(BaseObject o) {
        if (Simulator.objects.contains(o)) {
            Simulator.log.warning("Did not create object " + o.getName() + " duplicate name already exists");
        }
        else {
            Simulator.log.finest("Creating object " + o.getName() + "(" + o.getClass().getName() + ")");
            Simulator.objects.addElement(o);
        }
    }

    /**
     * Add an even to the Simulation.
     * @param e the event to add.
     */
    public static void addEvent(BaseEvent e) {
        Simulator.log.finest("Adding event " + e + "(" + e.getClass().getName() + ")");
        Simulator.events.addElement(e);
    }

    /**
     * Add an action to the Simulation
     * @param a the actoin to add.
     */
    public static void addAction(BaseAction a) {
        Simulator.log.finest("Creating action " + a.getClass().getName());
        if (!Simulator.actions.contains(a)) {
            Simulator.actions.addElement(a);
        }
    }

    /**
     * Check if we can create provided actions from our current state.
     * @param myclass the action to be checked.
     */
    private static void checkAction(Class<?> myclass) {
        Constructor<?> constructors[] = myclass.getConstructors();
        for (int i = 0; i < constructors.length; i++) {
            Simulator.testActions(constructors[i]);
        }
    }

    /**
     * Test all possible combinations with an action specified by the passed constructor.
     * Only objects from the current state of the simulator are used.
     * @param c a constructor of an action
     */
    private static void testActions(Constructor<?> c) {
        // One vector containing X vectors with objects where X is the number of arguments needed to construct this action.
        Vector<Vector<Object>> parameters = new Vector<Vector<Object>>(c.getParameterTypes().length);
        int i = 0; // this int is reused multiple times in this method
        // Initialize an empty vector for each parameter.
        for (i = 0; i < c.getParameterTypes().length; i++) {
            parameters.add(i, new Vector<Object>());
        }
        //loop through all objects
        i = 0;
        for (BaseObject o : Simulator.objects) {
            //check if the classtype fits one of the classes
            i = 0;
            for (Class<?> cl : c.getParameterTypes()) {
                if (cl.isInstance(o)) {
                    // add object to the associated list
                    parameters.get(i).addElement(o);
                }
                i++;
            }
        }
        // If there is an empty vector this means this action cannot be created.
        for (Vector<Object> v : parameters) {
            if (v.isEmpty()) return;
        }
        // Use recursion to create all possible combinations of the objects in the vectors.
        recursiveCheck(new Vector<Object>(parameters.size()), parameters, c);
    }

    /**
     * Loop through all objects that can be passed as first argument
     * Check if you can create an action object with the vector of arguments, if not, add (all possible) next arguments and
recursively try again.
     * @param v the vector with arguments (this is the only object that changes during recursion).
     * @param parameters a vector with vectors, each argument has a vector of possible objects.
     * @param c the constructor of the action to create.
     */
    private static void recursiveCheck(Vector<Object> v, Vector<Vector<Object>> parameters, Constructor<?> c) {
        //Check if the vector with elements is filled. (iow: we have enough elements to create an action object)
        //FIXME: Maybe some performance could be won by using a more efficient duplicate check
        // Check for duplicates: HashSet does not allow duplicates, skip all next iterations if a duplicate is found.
        HashSet s = new HashSet(v);
        if (s.size() != v.size()) {
            return;
        }

        if (v.size() == parameters.size()) {
            //try to create our object
            try {
                // Create the action object using the parameters from our vector v.
                BaseAction action = (BaseAction) c.newInstance(v.toArray());
                //System.out.println(action.enabledStatus());
            }
            catch (Exception e) {
                e.printStackTrace();
            }
            //we are done! we return to the previous invocation.
            return;

        }
        //Loop through all possible parameters we can use as next argument
        for (Object o : parameters.get(v.size())) {
            // add the next element to the vector
            v.addElement(o);
            // pass the vector to check again
            Simulator.recursiveCheck(v, parameters, c);
            // when we return here, our element has been checked. remove it and try the next
            v.removeElementAt(v.size() - 1);
        }
    }

    /**
     * Get an array of classes in a specific package.
     * @param pckgname the name of the package
     * @return
     * @throws ClassNotFoundException if no classes are found or an error occurred.
     */
    private static Class<?>[] getClasses(String pckgname) throws ClassNotFoundException {
        ArrayList<Class<?>> classes = new ArrayList<Class<?>>();
        // Get a File object for the package
        File directory = null;
        try {
            ClassLoader cld = Thread.currentThread().getContextClassLoader();
            if (cld == null) {
                throw new ClassNotFoundException("Can't get class loader.");
            }
            String path = pckgname.replace('.', '/');
            URL resource = cld.getResource(path);
            if (resource == null) {
                throw new ClassNotFoundException("No resource for " + path);
            }
            directory = new File(resource.getFile());
        }
        catch (NullPointerException x) {
            throw new ClassNotFoundException(pckgname + " (" + directory
                    + ") does not appear to be a valid package");
        }
        if (directory.exists()) {
            // Get the list of the files contained in the package
            String[] files = directory.list();
            for (int i = 0; i < files.length; i++) {
                // we are only interested in .class files
                if (files[i].endsWith(".class")) {
                    // removes the .class extension
                    classes.add(Class.forName(pckgname + '.'
                            + files[i].substring(0, files[i].length() - 6)));
                }
            }
        }
        else {
            throw new ClassNotFoundException(pckgname + " does not appear to be a valid package");
        }
        Class<?>[] classesA = new Class[classes.size()];
        classes.toArray(classesA);
        return classesA;
    }

    /**
           * @return   A vector containing all the objects currently loaded in the Simulator.
           */
    public static Vector<BaseObject> getObjects() {
        return Simulator.objects;
    }

    /**
           * @return   A vector containing all the actions currently loaded in the Simulator.
           */
    public static Vector<BaseAction> getActions() {
        return Simulator.actions;
    }

    /**
     * @return the letter S followed by a number. This number is unique for the current state.
     */
    public static String getState() {
        String s = "";
        for (BaseObject o : objects) {
            s += o.hashCode();
        }
        for (BaseEvent e : events) {
            s += e.hashCode();
        }
        for (BaseAction a : actions) {
            s += a.hashCode();
        }
        if (!Simulator.statespace.containsKey(s.hashCode())) {
            Simulator.statespace.put(s.hashCode(), "S" + stateId++);
        }
        return Simulator.statespace.get(s.hashCode());
    }
}
```

## GSDL Simulator interface screenshots

The objects of a scenario and their corresponding HSM models.

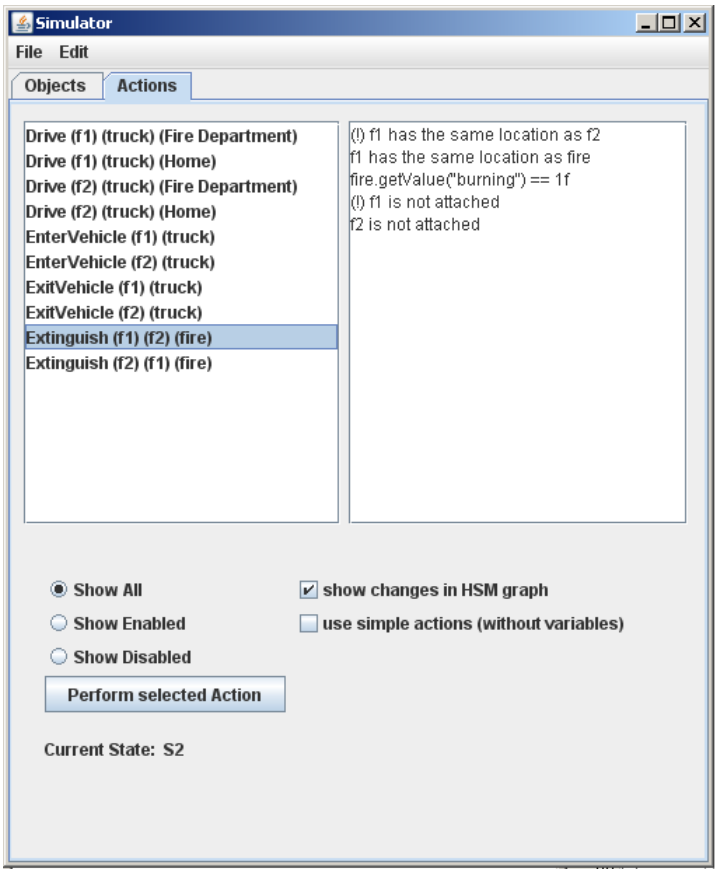


Location - Fire Department
Location - Home
FireMan - f1
FireMan - f2
FireTruck - truck
Fire - fire

f1
location is Home
attached to truck
relation inside truck

Figure 31 – GSDL Simulator screenshot including HSM models

The actions tab of the GSDL simulator interface. We can see why it is not possible for f1 to initiate the Extinguish action using f2 and fire. F1 is not at the same location as f2 and f1 is still attached to an object.



Drive (f1) (truck) (Fire Department)
Drive (f1) (truck) (Home)
Drive (f2) (truck) (Fire Department)
Drive (f2) (truck) (Home)
EnterVehicle (f1) (truck)
EnterVehicle (f2) (truck)
ExitVehicle (f1) (truck)
ExitVehicle (f2) (truck)
Extinguish (f1) (f2) (fire)
Extinguish (f2) (f1) (fire)

```
(!) f1 has the same location as f2
f1 has the same location as fire
fire.getValue("burning") == 1f
(!) f1 is not attached
f2 is not attached
```

● Show All
○ Show Enabled
○ Show Disabled

☑ show changes in HSM graph
□ use simple actions (without variables)

Current State: S2

Figure 32 - GSDL Simulator actions interface