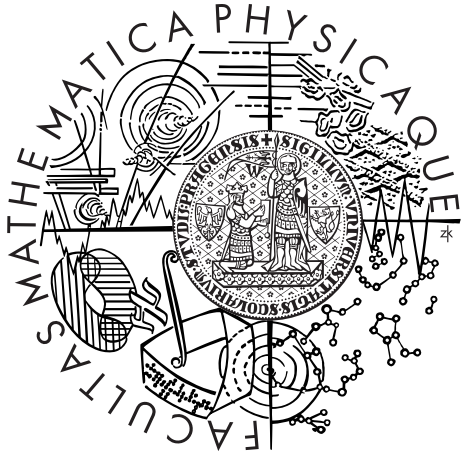
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



František Farka

Implementation of the SF-HRP action selection mechanism

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: Mgr. Tomáš Plch
Study programme: Computer Science
Specialization: IOI

Prague 2011

I would like to thank my supervisor, Mgr. Tomáš Plch, for his guidance and insight. I also would like to thank my family for motivation and support.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague 4. 8. 2011

Název práce: Implementace SF-HRP mechanismu výběru akcí

Autor: František Farka

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Tomáš Plch, Kabinet software a výuky informatiky

Abstrakt: Tato práce se zabývá implementací mechanismu State-Full Hierarchical Reactive Planning (SF-HRP) pro výběr akcí umělé bytosti v jazyce C++. Tato implementace je připojena k 3D virtuálnímu prostředí a umožňuje své profilování aplikacemi třetích stran. Prototyp takového profileru je součástí implementace. Práce také představuje vstupí formát pro popis chování agenta, který implementace používá. Implementace i vstupní formát jsou demonstrovány na testovacích scénářích. Koncept SF-HRP je zhodnocen vzhledem k obtížnosti návrhu chování umělé bytosti a vzhledem ke složitosti implementace.

Klíčová slova: SF-HRP, výběr akce, inteligentní umělé bytosti, reaktivní plánování, C++

Title: Implementation of the SF-HRP action selection mechanism

Author: František Farka

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Tomáš Plch, Department of Software and Computer Science Education

Abstract: In this thesis, we present our C++ implementation of the State-Full Hierarchical Reactive Planning (SF-HRP) mechanism for action selection for virtual agents. The implementation is connected to 3D virtual environment and provides access to 3rd party software for profiling purposes vie defined interface. A prototype of such a profiler is part of the implementation. The thesis also presents an input format for agent's behavior description and is used within the implementation. Both the implementation and input format are demonstrated on testing scenarios. The SF-HRP concept is discussed with respect to the difficulty of design of agent's behavior and complexity of the implementation.

Keywords: SF-HRP, action selection, intelligent virtual agents, reactive planning, C++

Contents

Introduction

An Artificial Intelligence (AI) has found extensive application in the entertainment industry. It has ever been important part of computer games, where for some of them the use of AI is vital. One example is the popular strategic life-simulation series The Sims[8], seen in Figure 1. Beside computer games, AI is also employed in other areas of the entertainment industry - movie industry [2] or virtual storytelling [9].

Nearly all of these applications are concerned with photo realistic visualisations, complex and dynamic 3D environments in order to increase realism. These virtual environments are populated by virtual agents which interact and tend to act human-like, having The Sims in mind. To maintain the illusion of a realistic world, agents have to act in a believable way, to be perceived intelligent [13].

Figure 1: Sims3 in-game screenshot

Some researchers consider intelligence to be a manifestation of behavior [13]. In simulated virtual worlds observers perceive agent's behavior through visualisation and agent's actions. An agent repeatably colliding with obstacles may be perceived as being unintelligent. On the other hand artificial agent may take actions human controlled agent would take in its place. Such an agent may be considered intelligent. Mimicking intelligence have been the main concern both in research and applications of virtual agents [2]. Based on [2] the solution to the problem of creating intelligent agent is agent with believable behavior, where agent's actions must be appropriate in the eyes of the observer.

The problem of selecting of an appropriate action from all available actions (to the agent in given moment) is done by *action selection mechanism* (ASM). Various ASM have been proposed, e. g. Internal Behaviour Network [11], LIDA architecture [12] and Reactive Planning [7].

Motivation

There are four known basic AI concepts [3]: first - acting humanly, second - thinking humanly, third - thinking rationally and fourth - acting rationally. Agent in the first concept, humanly acting agent, is an agent which chooses actions a real human in his place would. In second, humanly thinking agents emulates

human cognitive processes. Rationally thinking agents in third concept deduce their actions by a process humans consider rational. Fourth, rationally acting agents act in a way considered rational.

The most suitable approach in above specified entertainment oriented environments seems to be "acting humanly", to mimic believable human behavior focusing on visualisation of agent's actions [2].

Visualisation of a complex virtual world is demanding on computational resources. There are not many resources left for executing ASM of the virtual agent. Therefore the timely fashion of ASM execution is one of our main concerns besides the believability of selected actions. Nor completely unresponsive agent nor agent which waits too much before taking next action would not be probably considered intelligent.

Reactive planning, proposed in [7] seems to meet both requirements on believability and on time fashion. Further studies [1] have shown lack of certain aspects of real-world being's behaviour, intentions, cleanup or transitional behaviour etc.

Discussion of these limitations in [2] introduced extension to reactive planning, adapting techniques of finite state machines, called *State-Full Hierarchical Reactive Planning* (SF-HRP). However only limited implementation in simple 2D world was presented.

Even when having ASM producing satisfactory actions in acceptable timely fashion we wished to address the issue of designing the virtual agent's behavior. With increasing complexity of virtual worlds and the amount of virtual agents deployed increases the complexity of the design of these agents. The above mentioned SF-HRP eases the design complexity of virtual agent as it allows decomposition of the agent's behavior into several separate phases [2]. We also believe possibility of profiling the behavior of the agent in virtual world would be contributory for the design.

Thesis goals

The main goal of the thesis is to implement the concept of SF-HRP utilizing C++ programing language and connect the implementation to a modern dynamic 3D virtual environment. We implement an ASM library and a wrapper for chosen environment. The implementation focuses on design of the architecture, speed and reliability.

We also propose an input file format for behavior description of the virtual agent driven by SF-HRP. We address the issue of prototyping agent in virtual world with interface that our library provides to external tools. Both input file format and prototyping interface are discussed with respect to the difficulty of the design of agent's behavior.

We present examples of agent's scenarios which can be tested in the virtual environment we have chosen. We discuss bottlenecks of the concept based on the scenarios and implementation experience.

Thesis Structure

This thesis is divided into chapters. The first chapter is devoted to the theoretical concept of *Simple Reactive Planning* (SRP), *Hierarchical Reactive Planning* (HRP) and SF-HRP, based upon [2].

The second chapter describes the architecture of our implementation of the SF-HRP action selection mechanism. We present the input format for our application, reason the choice of virtual world used with our application and describe details of the implementation.

The third chapter present model scenarios. Based on these we discuss features of SF-HRP concept and our implementation. After the third chapter is included conclusion and future work.

The Appendix A contains Document Type Definition (DTD) of XML input format of our implementation. The Appendix B contains input behavioural files of scenarios discussed in Chapter 3. The Appendix C informs on installation process of our implementation.

1. State-Full Hierarchical Reactive Planning

Based on [2] we present our understanding of the concept of *Simple Reactive Planning* (SRP) and *Hierarchical Reactive Planning* (HRP). We take a closer look at it's limitations and corresponding improvements. We describe SF-HRP being an extension of HRP by means of finite state machines and present our understanding of every phase of agent's plan.

1.1 Reactive Planning

Humans tend to see a virtual agent's behavior as intelligent when the agent chooses actions which are similar to a real world being's choices in corresponding situation. Essential part of the believability of the agent's behavior is timely fashion of action selection [2].

In [3] planning is understood as process of creating a plan, sequence of actions, in order to achieve certain goal. When plan is rendered invalid, e. g. goal changes, planned action cannot be executed or for any other reason, which may happen relatively often in dynamic environments it needs to be recomputed. This is problematic as creation of plan is resource demanding process [2]. The alternative to plan precomputation has been presented [7] - the concept of *Reactive Planning*.

Reactive Planning lacks any sequence of actions scheduled for execution, selected action depends solely on state of the virtual environment. The ASM is based on condition execution decision making, where conditions - boolean expressions - represent context of the world. This approach renders shorter reaction times of virtual agents than planning with lookahead as there is no need for plan recomputation thus we believe it is good choice for the ASM in dynamic environments. We will describe the concept of *Simple Reactive Planning* and its extension *Hierarchical Reactive Planning*, both proposed in [7].

1.1.1 Simple Reactive Planning

Simple Reactive Planning (SRP) is flat, one level architecture. The key structure, referred to as a *basic reactive plan*, is a set of *condition-action* rules that can be formalised as a triplet

(PRIORITY, CONDITION, ACTION)

where

priority specifies rule importance

condition is a boolean expression

action is an atomic action or sequence of such actions

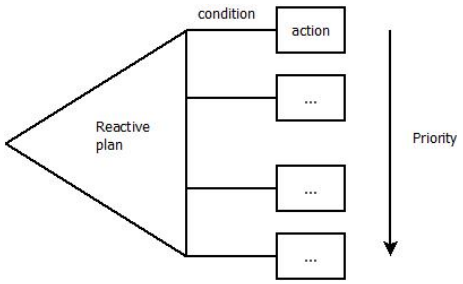


Figure 1.1: SRP plan diagram

The *action-condition rule* is called simply *rule* and *simple reactive plan* (Figure 1.1) just *plan* where there is no ambiguity. The goal of the ASM is to select rule with the highest priority having a holding releaser for execution. This can be effectively done by checking rules in descending order by priority which gives Algorithm 1.

Algorithm 1 SRP ASM algorithm

```
procedure ASM                                   ▷ SRP action selection
    for all rules as rule in descending order by priority do
        if condition of rule is satisfied then
            ExecuteAction(action of rule)
        end if
    end for
    ExecuteAction(idle action)
end procedure
```

The rules are called:

active when being executed

preactive when its release condition is satisfied, but is not active, i. e. there is another rule with satisfied release condition and higher priority

inactive when condition evaluates false

suspended when rule was surpassed in execution, i. e. when rule was active and another rule became active

1.1.2 Hierarchical Reactive Planning

Hierarchical Reactive Planning (HRP) is an extension to SRP. It allows simple actions and action sequences to be replaced by expansion into plans, thus creating a tree like structure with action primitives as leafs and HRP plans as inner nodes, called *behavioural tree* (*be-tree*) [2] as seen on Figure 1.2. In following text the set of HRP plans with same distance from root plan is called a *level* and the root node of i called *top-level*. Action primitive may be both an atomic action and an action sequence.

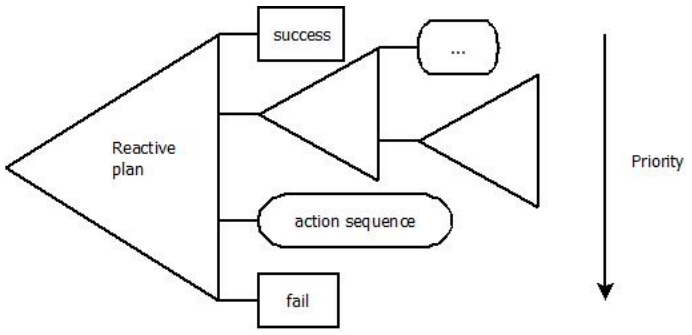


Figure 1.2: HRP plan diagram

This enforces change of an ASM Algorithm 1 into a recursive algorithm. When ASM chooses a plan as a candidate rule for execution, it travels down the tree and continues evaluation on next level until it hits leaf rule.

Another addition to SRP is an explicit success and fail of the plan and its propagation to upper levels of the be-tree. Success/fail is considered a special atomic action. Execution of this action forces ASM to continue selection in parent node.

Introducing hierarchical behavior to the ASM allows designer to create plans with top-level based behavior. Using this concept plan can be divided into certain tasks, e. g. our plan for morning could be "brush the teeth", "carry out the trash", "catch the bus" [1].

These modifications of SRP gives Algorithm 2. The idea of the algorithm is to select a top level plan and then travel down the be-tree's structure until reading an atomic action (being leaf of the be-tree) which is executed.

1.1.3 Improvements of Hierarchical Reactive Planning

Problems arising from the discussed ASM were addressed in [1] and [2]. We describe our understanding of the presented problems and corresponding solutions. All solutions are designed with the idea of improved believability with low computational cost in mind.

Interrupt safe rules

Some patterns in agent behavior may consist of more than one atomic action, but still need to be executed in atomic fashion. For this reason [2] introduces an interrupt-safe flag. Rules marked with this flag cannot be surpassed by another rule. This helps the agent to keep consistent, more believable behavior. Such action sequences should be kept short not to render agent non-responding to the environment.

Releaser safe rules

When constructing HRP rules, designer may reach a situation that he intend to use a rule where an action sequence influences it's releaser in such a way the

[1] Of course we design "carry out the trash" rule releaser so that when under time pressure, it renders false and we rather catch the bus and do not care for trash.

Algorithm 2 HRP ASM algorithm

```
procedure ASM                                          ▷ top-level plan selection
    active_plan ← plan with holding releaser and highest pririty
    if no active_plan then
        (ExecuteAction)idle action
    end if
    if priority of active_plan > priority of previously_executed_plan then
        PlanAS(active_plan)
    else
        PlanAS(previously_executed_plan)
    end if
end procedure

procedure PlanAS                                       ▷ active rule selection
    active_rule ← rule in plan with holding releaser and highest prority
    if no active_rule then
        Fail
    end if
    if priority of active_rule > priority of previously_executed_rule then
        ExecuteAction(action of active_rule)
    else
        ExecuteAction(action of previously_executed_rule)
    end if
end procedure

procedure ExecuteAction(action)
    if action is a plan then
        PlanAS(action)
    else if action is an atomic action then
        Perform(action)
    else                                   ▷ action is a sequence of atomic actions
        atomic_action ← next atomic action in sequence
        if no atomic_action then
            Perform(first atomic action in sequence)
        else
            Perform(atomic_action)
        end if
    end if
end procedure
```

releaser ceased to hold. The action sequence is obviously interrupted in-between its atomic actions. For this reason [2] introduced releaser-save flag that keeps rule in execution, until surpassed by higher priority rule, even if the releaser condition is not satisfied anymore.

Sticky rules

Some rules may be designed to react to high-priority events, but still not to become active in case of a holding releaser (e. g. interrupt-safe rule is in execution). In [2] were proposed sticky flag and a timeout to keep a rule in preactive state longer, but not indefinitely. It means the sticky rule is given some time as candidate rule for execution after the currently executed rule stops execution, but only up to timeout.

1.2 State-Full extension to HRP

Specific problems and limitations of HRP approach and corresponding solutions were discussed in [2]. The basic argument was that SRP and similarly HRP, even if it has tree structure, is a simply ordered set of condition-action rules and lacks any explicit state. This complicates both internal control of information flow by the ASM and external analysis of plan execution, either manual or automated. Humans tend to certain decomposition of real world plans into different phases. They usually need to meet some prerequisites at first, then cope with the task itself and finally do some clean up, not taking any interruptions in account.

In the light of these aspects [2] seemed contributory to introduce finite state machine extension to HRP in order to help the designer create the plan and better inspect its execution. In following section we would like to present our understanding of this idea.

States of a proposed finite state machine constructed above the original HRP, each with a specific purpose in plan execution, are:

1. **initialization** - plan is prepared for execution
2. **execution** - original HRP ASM
3. **termination** - execution has finished and terminating behavior is executed
4. **cleanup** - gathered objects are cleaned up etc.
5. **finish** - plan is not active anymore
6. **switch out** - plan is preparing to suspend its execution
7. **switched** - plan is suspended
8. **switch in** - plan is restoring from switched state
9. **emergency** - emergency situation has occurred

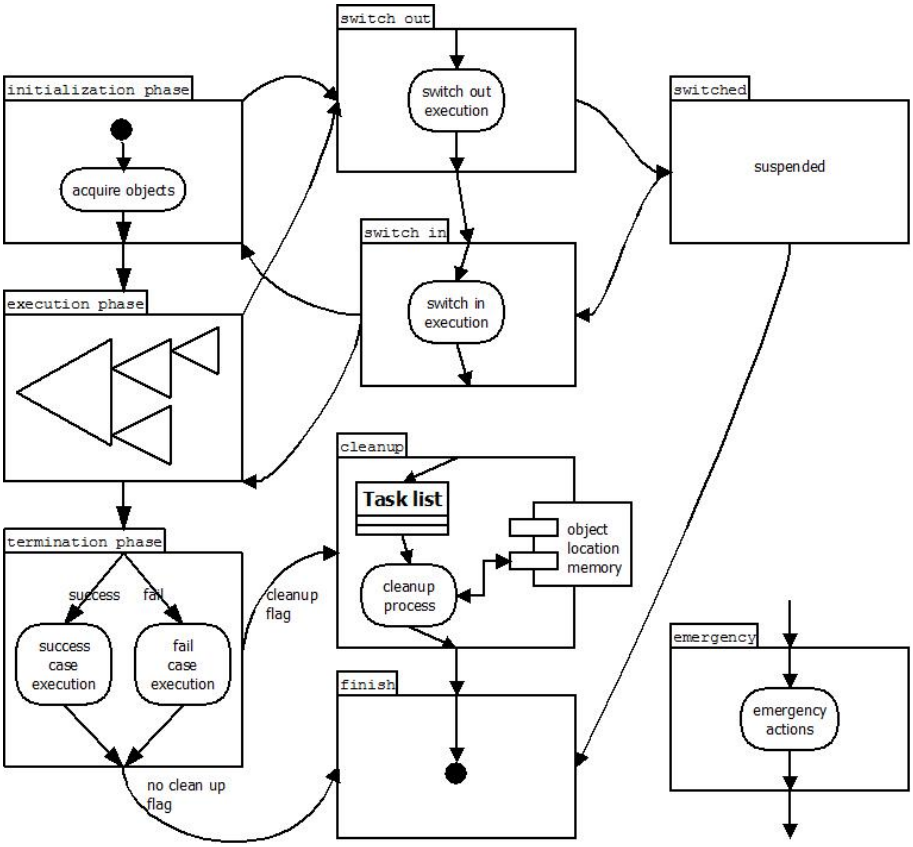


Figure 1.3: state diagram of SF-HRP

This mechanism restricts rather chaotic[2] execution of a plan only to it's execution state (or phase). The state transition function is presented in form of state diagram in Figure 1.3 for we believe it is easier to understand. Initial state is *initialization* and the only final state is *finish*. States of diagram are properly described below, transitions are displayed as arrows.

The concept also copes with transitional behavior via plan switching. When another plan with higher priority gets active current plan is supposed to perform actions in order to get to consistent, defined state, when in *switch out* state, and then suspended. Similarly it is recovered from such a state when switched in. In the text we use the term *suspender* for plan which is being switched in, the term *suspendee* for plan which is being switched out and the term *suspended plan* for plan in switched phase, i. e. plan which has been successfully switched out.

1.2.1 Initialization phase

Typical reactive plan requires some objects for its successful execution. Acquiring of these items is handled in HRP by high priority rules - initiatory rules. SF-HRP introduces initialization phase as replacement of these rules. The benefits of this approach are lower complexity and better consistency[3] of resulting plan.

[2]from designer's point of view

[3]Execution phase is entered either when all items are gathered, thus in consistent state, or is not ever entered. Initialization can be also used as fallback in case of some item is lost during execution.

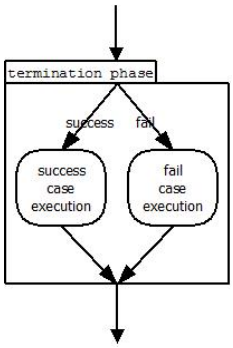


Figure 1.4: termination phase

Objects used by plan are managed via object sets and object-class sets, in order to avoid problems with explicitly specified objects and provide more flexibility, both discussed in [2].

Object set

Object set concept introduces two object containers, object list and object set. The first - an object list- is a list of items that are all required, thus list is considered satisfied when all are found. Second is object set itself as ordered set of object list ordered by descending importance. Object set is satisfied, when any of lists is satisfied, thus objects are being satisfied by their importance. The object set is viewed as a boolean formula in disjunctive normal form, disjunction of object lists, each object list as conjunction of objects.

Object class set

Similar semantics as for object set is used for class set, only the object is replaced with object class construct. Specification of an explicit object is avoided, declaring only traits and/or purpose of required object is preferred, allowing agent to acquire object that satisfies its needs best, e. g. are closest, most easily obtainable, etc.

1.2.2 Execution phase

The execution phase consists of original HRP concept with modifications presented in 1.1.3, as interruption safe flags or sticky rules. Modifications of HRP enforces changes in its structure, discussed in section 1.1.3, and ASM algorithm, presented in Algorithm 3.

1.2.3 Termination phase

The termination phase is introduced in order to perform some actions based upon result of execution phase. Agents may present different behavior in case of success or fail of the plan, get angry or feel happy (Figure 1.4).

Termination phase is followed by finish phase.

Algorithm 3 SF-HRP ASM algorithm

```
procedure ExecutionAS                          ▷ execution phase action selection
    if actual_rule is a interrupt-safe then
        selected_rule ← actual_rule
    else
        candidate_rule ← rule with highest priority considering rules
        with active releaser and sticky rules within timeout

        if actual_rule is releaser-safe and its releaser is false then
            if priority of actual_rule > priority of candidate_rule then
                selected_rule ← actual_rule
            else
                selected_rule ← candidate_rule
            end if
        else
            selected_rule ← candidate_rule
        end if
    end if

    Execute(candidate_rule)
end procedure
```

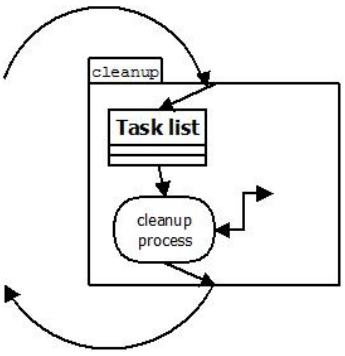


Figure 1.5: cleanup phase

1.2.4 Cleanup phase

The cleanup phase is important part of human behavior. As [2] introduces it, it has a reverse structure to the initialization phase only with slight differences. In initialization phase agent may gather more items than it necessary needs[4]. On the contrary agent may omit some items needed by other plans from cleanup. Structure of cleanup phase is presented on Figure 1.5. After the cleanup, plan transits to finish state.

In the case of externally induced fail of a plan cleanup phase is never performed, possibly leaving such plan with object residues. Parent plans are made [2] responsible for those objects and supposed to perform appropriate cleanup.

[4]Due to object sets and object class sets

1.2.5 Finish phase

Finish phase is final phase signalling the plan has finished its execution. When marked with cleanup flag, cleanup phase is executed at first. This is contributory both for the plan's designer, as he can design cleanup behavior either in a separate phase or in a HRP plan[5], and programer, as he can easily perform engine specific tasks (resource management etc.).

1.2.6 Switch out

When suspender, another plan with higher priority, gets active, current plan needs to be suspended. In order to ensure consistency in agents behavior and also plan execution, suspendee has special phase, switch out phase, that allows it to perform some actions and leave agent in consistent state so suspender can smoothly start its execution (e. g. agent does not hold any items etc.).

1.2.7 Switched

After the plan performs switch out actions, it renders switched and performs no actions. Later when suspender stops its execution successfully, suspendee resumes its execution and continues to switch in phase. When suspender fails, suspendee continues directly to finish phase (i. e. no cleanup is performed).

1.2.8 Switch in

When resuming from switched phase, suspendee needs to recover from the *switched* state, for example put some items into agent hands. Switch in phase may follow directly after switch out plan in case that switching suspender's releaser ceased to hold.

1.2.9 Emergency

Emergency phase is special phase and is entered when exceptional circumstances occurs. The phase is performed immediately, thus can model reflex-like behaviour. After emergency actions are executed, plan continues in its former state.

[5]Or does not design it at all

2. Implementation

We developed a configurable State-Full Hierarchical Reactive Planning Action Selection Mechanism (SF-HRP ASM) connected to complex 3D environment utilizing the C++ programing language. In this chapter we discuss an architecture and implementation details. We also elaborate on a choice of connected virtual environment. We mention a XML input format specifying agents behavior and describe interface which our implementation provides for external inspection of SF-HRP plan execution.

2.1 Architecture

The goal of our implementation is to test the SF-HRP concept in complex virtual environment with respect to the believability and easy design of agent's behavior and discuss features and bottlenecks of the concept. We identify four preconditions to achieve the goal.

(a) implementation of SF-HRP ASM configurable with different agent behaviours

(b) virtual environment in which the agent takes its actions

(c) profiling tool for feedback on agent's behavior in the environment to the designer

(d) a way of communication among those three parts.

Given the specified preconditions we address following requirements:

1. **input format for behaviour specification** which is understandable for designer either directly or with a tool
2. **input format parser** which collaborates with ASM
3. **ASM implementation** which understands the input behavioural format
4. **choice of virtual environment** in form of game engine which is possible to utilise
5. **profiling tool** informing on execution of agent's behavior
6. **wrapper** managing communication among the virtual environment, the ASM and the prototyping tool

The requirements 1 to 3 satisfy the precondition (a) and the rest of requirements satisfy the preconditions 4 to 6 in given order.

We have designed a XML format for behavior specification. The choice of a XML technology, structure of the format and design and implementation of the parser are described in Section 2.2 and 2.3 respectively.

The ASM implementation consists of set of mutually interconnected primitives, each primitive reflecting a primitive of SF-HRP concept, e. g. phase

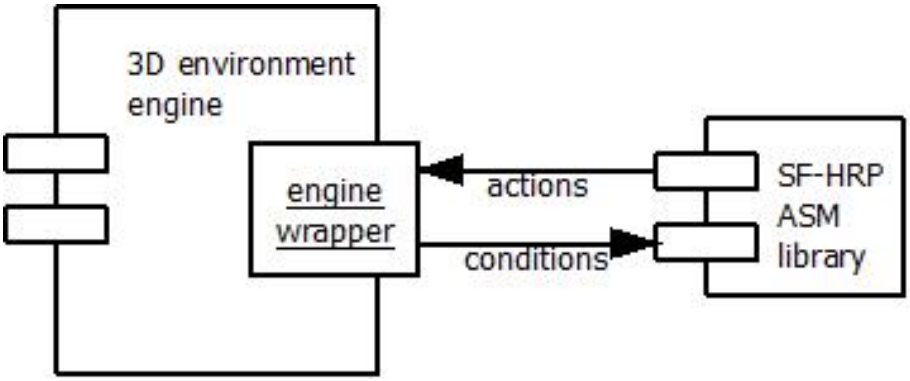


Figure 2.1: Architecture of the implementation

of SF-HRP plan, plan rule, condition. Implementation is properly described in Section 2.4.

Both parser and ASM are designed to be independent on chosen engine. It means their implementation is not dependent on engine architecture and implementation, nor actions nor conditions of engine are hard coded directly into the ASM or parser. This is particularly due to the fact main part of the ASM were designed before exact engine was chosen and also due to the fact we like the idea of ASM independent on engine and possibility of engine interchanging, so our ASM works under different engines without major modifications.

Because the ASM and the parser are independent on the rest of the implementation and because we prefer easy change of a connected engine we decided to encapsulate them into a separated subsystem - the library - as shown on Figure 2.1 which communicates with rest of the implementation through defined interface. We call the library *SF-HRP library*.

The choice of connected virtual environment - the engine - is discussed in Section 2.5. The related wrapper utilising engine internal structures, configuration files and maps used for testing is described in Section 2.6. Wrapper layer is the only part of our project which needs to be reimplemented in case of use with different game engine.

We provide only a simple implementation of profiler described in Section 2.7 which is closely related to the engine. An implementation of more suitable profiler that eases design of agent's behavior is beyond the scope of this thesis.

2.2 Input format for behaviour specification

The input format is proposed in order to ease the design of agents behavior. We have chosen textual format over binary for it is directly readable by designer. We prefer XML over any other standard or self-designed format for it is self-descriptive thus easier to understand and its specification is freely available at [14]. Document Type Definition (DTD) is included in Appendix A.

The structure of XML format is designed in order to reflect the concept of SF-HRP ASM described closely in Chapter 1. Agent is driven by set of SF-HRP plans, each plan triggered by condition. SF-HRP plan consists of separate phases - initialisation, execution, termination, cleanup, etc. The execution phase, containing most of the agent's behavior, consist of tree structure of *simple plan*s.

Each *simple plan* contains rules. Among rules is selected agent's action according to their properties, i. e. priority, weight, condition, action, flags etc. The structure of SF-HRP ASM concept is tree-like which corresponds to the tree structure of well-formed XML document.

We describe the structure on examples of scenarios discussed in Section 3.2 included in Appendix B. Document entity is called *agent*. Each children element called *sfhrplan* specify one SF-HRP plan of the agent. Given the example the basic scheme of the document (with one SF-HRP plan) is:

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE agent SYSTEM "sfhrp.dtd">
<agent>
  <sfhrplan>
    ...
  </sfhrplan>
</agent>
```

The *sfhrp* element consists of element *cond* describing triggering condition and elements describing separate phases of SF-HRP plan. The *init* element for initialization phase, the *hrplan* element for execution phase, the *term* element for a termination phase and the *cleanup* for clenaup phase. Finish phase is not included as it does not require any modifications by designer. Thus the structure of *sfhrp* element is:

```
  <sfhrplan>
    <cond id="plan-state-full">
      ...
    </cond>
    <init>
      ...
    </init>
    <hrplan entry="onlyone">
      ...
    </hrplan>
    <term>
    </term>
    <cleanup enabled="true" />
  </sfhrplan>
```

2.2.1 Conditions

A condition is described by *cond* element. The element optionally contains attribute *id* which serves for condition identification during profiling. The children elements of *cond* element are elments *elem* for atomic condition used by engine (2.5), *and* for conjunction, *or* for disjuntion and *not* for negation. Atomic condition *elem* consist of type of the condition specified by element *type* and zero or more parameters, each specified by element *param*. Condition is interpreted in postfix order. One of the conditions used in examples is:

```
<cond id="dog-has-not-ball">
  <elem>
    <type>COND_DOG_HAS_BALL</type>
  </elem>
  <not />
```

```
    <elem>
      <type>COND_BALL_IN_AIR</type>
    </elem>
    <and />
  </cond>
```

The condition renders true when an agent "dog" has not an object "ball" nor is waiting for it as the ball is in air.

2.2.2 Initialization phase

An initialization phase contains list of objects to be gathered by agent. The set is described by *list* element containing one or more *object* element, specifying the object. Object list may be omitted. Initialization phase in which is agent supposed to gather one object identified as "ball" is described as:

```
<init>
  <list>
    <object>ball</object>
  </list>
</init>
```

2.2.3 Execution phase

An execution phase is described by element *hrplan*. The element contains multiple children elements *plan* and attribute *entry* which is required and specifies root element of *simple plan* tree. The *plan* element describes one *simple plan* and has required attribute *id* which is used for referring to the plan.

```
<hrplan entry="onlyone">
  <plan id="onlyone">
    ...
  </plan>
</hrplan>
```

The *plan* element consist of multiple *rule* elements, each specifying single plan rule. Rule is described with elements *prio* for priority, *weight* for weight, *cond* for condition described in Section 2.2.1, *exec* for action, *flags* for rule flags and *sttou* for sticky flag timout. Rule has optional attribute *id* used for identification with profiler.

Element *exec* describing triggered action has attribute *type* which is required. Attribute specifies explicit success and fail rules, plan rule and action rule. Actions within action rule are specified by multiple elements *action*. The element action has children elements *type* and *params* with same semantics as in condition element *elem* in Section 2.2.1. The example of plan with action rule and explicit success rule is:

```
<plan id="onlyone">
  <rule id="rule-throw">
    <prio>3</prio>
    <weight>1</weight>
    <cond id="dog-has-ball">
      ...
```

```
      </cond>
      <flags></flags>
      <exec type="action">
        <action>
          <type>ACTION_DOG_THROW_BALL</type>
        </action>
      </exec>
    </rule>
    <rule id="rule-succ">
      ...
      <exec type="success" />
    </rule>
    ...
  </plan>
```

The flags are specified with children elements *sticky* for sticky flag, *relsafe* for releaser-safe flag and *intersafe* for interruption-safe flag.

2.2.4 Termination phase

A termination phase has two children elements. In case the execution phase ends with success then an action specified in *succ* element is executed, in other case an action in *fail* element is executed. Termination phase element may be omitted.

2.2.5 Cleanup phase

A Cleanup phase element signals whether the cleanup is performed after execution phase. It is specified in attribute *enabled*.

2.3 Input format parser

The parser reads input file in XML format discussed and specified in Section 2.2 which describes behaviour of a virtual agent and produces ASM from primitives described in Section 2.4. Parser creates and initialises these primitives.

We use *libxml2* [6] library and *Simple API for XML 2 (SAX2)* for parsing. We have chosen this library for several reasons. At first it is well documented at [6]. Next reason is *libxml2* is distributed under MIT license which allows easy integration with different licenses. Finally *libxml2* integrates well with C++ which is implementation language of our application. We use *SAX2* API over *Document Object Model (DOM)* for we do not need to traverse the document tree. The input format (2.2) is designed in such a manor sequential reading of input is sufficient.

Parser consist of three main parts. These are parser state, parser logic based on state of parser and factory which provides interface for use of the parser from outside the library. We describe these parts separately.

Parser state

Parser state in form of object *CParserState* is passed by *libxml2* library to our callbacks for SAX2 API. Parser state holds information on state of the parser

itself, information for generating ASM primitives, e. g. string identifiers and plan priorities, and created and particularly created primitives, e. g. plans and conditions, for further processing and integration into ASM.

Parser logic

As mentioned above, we use *SAX2* for purposes of our library. This API uses callbacks for different events occurring when input file is parsed by *libxml2* [6]. We use callbacks for a starting element, an ending element and characters within an element. These callbacks are implemented as static members of *CPlanParser* object. This object also provides error handling for errors connected with input file format.

ASM Factory

Interface to the parser for use from outside the library is provided by means of factory object *CSFHRPFactory*. This interface is described in 2.4.1. The factory object initialises parsing, calls *libxml2* service and propagates constructed ASM or reports error from out of the library to the user of the library.

2.4 ASM Implementation

The SF-HRP ASM is constructed by parser, described in 2.3. The implemented ASM has hierarchical structure, corresponding to structure of SF-HRP described in chapter 1.

The top level contains set of SF-HRP plans and select active plan with regard to the holding condition, priority and plan switching behavior. This is the only level of ASM which ought to be manipulated directly by user of the library.

SF-HRP plan consists of primitives for different phases (eg. for initialization, execution, cleanup, ...) and selection among these based on figure 3. Plan also holds its triggering condition and propagates information on condition activation and deactivation to the upper level. The most important are initialization, cleanup and execution phase. Both initialization and cleanup communicates with engine and manages manipulation with objects used by agent. Execution phase represents original HRP of which is SF-HRP extension.

The execution phase - original reactive plan - is implemented as tree structure of plan nodes and action nodes. Plan nodes are inner nodes of the tree and represent children plans. Plan nodes propagate action selection between levels of the tree with respect to sticky, interruption safe and releaser safe flags. Action nodes create leaf nodes of the three. In action nodes is selected actual action. Both action and plan nodes are triggered by condition. From implementation point of view these conditions are the same conditiona which are used in case of SF-HRP plans, mentioned above.

The ASM is implemented as set of primitives mutually linked together. Each primitive is represented by C++ object. We describe the most important of these objects further.

SF-HRP ASM

On top of the ASM hierarchy is object of *CSFHRPASM* class returned by factory *CSFHRPFactory*, described in 2.3. This object realises plan selection according to the top level plan selection algorithm [2]. Condition subsystem used for plan releasers is common to multiple classes and is discussed below. *CSFHRPASM* stores SF-HRP plans and is responsible for their destruction.

SF-HRP plan

SF-HRP plan represents state machine extension to the original HRP concept. We implement this plan logic with *CSFHRPPlan* class, every SF-HRP plan of agent is instantiated as an object of this class. This object is mainly responsible for HRP plan selection acording to Algorithm 3. It delegates request for action code to appropriate phase of plan, every phase represented by single object. We discuss notable object/phases further in text below. We use dedicated object for every phase as it allows to change behavior of agent in certain state of the plan only by reimplementation of this object.

Initialization phase

The object representing initialisation phase communicates with connected virtual world via *IEngineDescriptor* interface. This object is responsible for acquisition of objects[1] used by agents. It cooperates with cleanup object in order to keep consistency in object acquisition and cleanup.

Cleanup phase

The object representing cleanup phase communicates with connected virtual world via *IEngineDescriptor* an initialization phase object. It is responsible for cleanup of objects acquired in cleanup phase.

Execution phase

Execution phase encapsulates original HRP plan behavior and is represented by object of *ReactivePlanner* class. Object contains recursive rooted tree structure of *SimplePlan*s and redirects requests for action code on it, with respect to the InterruptSafe, ReleaserSafe and Sticky flags. This object also stores pointers to all *SimplePlan*s in tree and manages their destruction. We prefer this implementation over recursive destruction of the tree as it allows us to share common subtrees on *SimplePlan* structure.

SimplePlan tree

Objects of *SimplePlan* serves as nodes of be-tree and containers for plan rules. Simple plan chooses its rule with holding releaser and highest priority and propagates action selection to it. Rules use the same condition subsystem for their releasers as SF-HRP plans. Condition structure is discussed below. Rules are derived from class *BaseRule* and are of two types:

[1] Objects in virtual world, e. g. virtual axe or ball.

ActionRule class object contains sequence of action codes obtained via *getAction()* method of *IEngineDescriptor* interface. This object is effectively on lowest position in whole structure of the SF-HRP ASM and represents leaf of the *be-tree*.

PlanRule class object basically contains pointer to another *SimplePlan* class object and passes action selection on it. This object represents internal node of *be-tree*.

Conditions

We use conditions in form of doubly linked rooted trees consisting of elementary conditions, which are understood and set to appropriate value by engine via wrapper and *SetCondition* method of ASM object, and boolean operators **and**, **or** and **not**. Elementary conditions represent leaves and operators inner nodes of condition tree.

We use doubly linked tree in order to minimise cost of reevaluation of condition tree. Single condition tree is reevaluated only when some of its elemental conditions change its value. Such a change is propagated into parent nodes and is stopped when does not change logical value of an internal node.

These condition trees are used both as top level releasers of SF-HRP plans and releasers of rules in HRP plans. There is no need for distinguishing between plan and rule conditions. It also ease construction of XML input file parser.

2.4.1 Public interface

Our library communicates with engine wrapper through defined interface in order to hide implementation details and restrict source code dependencies. We describe this interface and its intended usage separately from library internal structure. Although classes in our library contain many public members, many of them are not intended for direct use from within the wrapper. We decided to leave this members public rather than implement them as private/protected and use C++ friend construct where necessary. This approach possibly allow further extensions to our library, e. g. online be-tree modifications mentioned in [2]. We does not mention these in description of public interface of our library. We also take closer look at implementation of input file parser and discuss choice of used library.

Creation on SF-HRP planner from given input file is complex, multiple step procedure as input file needs to be parsed and different objects of plan dynamically instantiated and linked together. Our library provides factory for this task:

```
class CSFHRPFactory {
public:
  static CSFHRPASM* CreateASM(
    std::string path,
    IEngineDescriptor* engineDescriptor,
    IInspector* inspector = null;
  );
};
```

We have to provide this factory with path to input file specifying agents behavior in XML format discussed below, engine descriptor object and ASM inspector object. Although our library is engine independent, it must negotiate some common context with engine it is collaborating with. For this reason it must be provided with object implementing *IEngineDescriptor* interface:

```
class IEngineDescriptor {
public:

  virtual int RegisterCondition(
    std::string type,
    std::vector< std::string> params
  ) = 0;

  virtual int RegisterAction(
    std::string type,
    std::vector< std::string> params
  ) = 0;

  virtual int GetIdleAction(
    std::string type,
    std::vector< std::string> params
  ) = 0;

};
```

Methods *RegisterConditon* and *RegisterAction* returns codes of conditions and actions based on type and parameter textual identifiers loaded from input file. *RegisterCondition* also informs engine which conditions does library expects to be informed of. *GetIdleAction* method provides code for idle action.

We use integer codes for conditions and actions in order to minimise data transfer between engine and our library during plan execution. Textual identifiers are necessary only when initializing an agent. This also allows certain abstraction of our SF-HRP ASM implementation which does not need understand nor conditions from engine nor actions of agent and lays engine/wrapper responsible for textual identifier representation thus those are not hard-coded in library.

Factory may be optionally provided with inspector object. This object is intended for propagation of information on changes of internal state of the planner, regardless who is receiver of this information. It may be both engine itself as in case of our implementation 2.3 displaying these changes in developer console or some third-party prototyping tool, e. g. Pogamut [10]. Expected informer object must implement following interface:

```
class IInspector {
public:

  virtual void
  processEvent(int, std::string) = 0;

};
```

Implementation of *IInspector* is provided with code of event and id of element in input file if available on which event occurred. We define following events:

COND_ACT when condition is activated

COND_DEACT when condition is deactivated

ACTION_SEL when action in action rule is selected for execution

RULESUCC_SEL when explicit success rule is selected for execution

RULEFAIL_SEL when explicit fail rule is selected for execution

User of the factory is responsible for destruction of returned object. Created SF-HRP planner has following public interface:

```
class CSFHRPASM {
public:

  void SetCondition(
    int condition,
    bool value
  );

  int GetAction();
};
```

Object of *CSFHRPAMS* expects engine to inform it of changes of elemental conditions via *SetCondition*. Object provides action code via *GetAction* on demand. Our library is deliberately designed in this manor not to restrict its use to certain engine architecture. Collaborating engine may require both synchronization of agent actions with its internal ticks or fetch agent action on demand[2] and it does not affects architecture of our library, only requires engine wrapper to correctly pass requests of engine to our code.

2.5 3D environment

The aim of our thesis is to test SF-HRP concept in much more complex and believable environment than the simple prototype in [2] is. We believe reasonable choice for such a task is modern 3D game engine with modding support. There are several game engines available, even with further development tools for easier manipulation with many source files which such a complex piece of software needs for its correct function, We were choosing between these development toolkits:

Source SDK and Source Engine by Valve Software [4], used in games Half Life 2, Left 4 Dead, Portal etc.

UDK, Unreal Developer Kit and Unreal Engine by Epic Games [5] used in games as Unreal Tournament 3, Gears of War or BioShock Infinite.

[2] Or even any other approach

Either of these two satisfies our demand on complex, visually attractive 3D world with wide range of agent actions as a result of many different signals from environment. Both are also quite well-documented by developer, with wide community of users on forums and mailing lists. They are equally written in the C++ programing language.

We have chosen Source SDK for following reasons:

- Unreal engine manages low level tasks as graphics, and use UnrealScript language for authoring gameplay events as AI. Source Engine has no such scripting layer and keeps this logic directly in compiled C++ source code, therefore we believe it is more suitable for fast-running AI implementation which is one of subjects of our thesis.

- Source Engine allows user to create mods[3] for existing Source game and deploys many of the contents in ready to use state. Unreal on the other hand is intended to create stand-alone games. As we do not intend to develop game but rather test particular ASM concept we believe Source Engine is more suitable as it leaves less unrelated game-development work in our hands.

2.6 Wrapper

We have chosen Source Engine from Valve Sofware as virtual environment of the implementation for reasons discussed in 2.5. Source SDK is distributed as binary executable, the engine itself, a set of tools generating source code project - source code of parts of game dedicated for modding - and tool working with used maps and models of which the Hammer Map Editor is of our interest. Every mod is based on existing game, using Source Engine, and provides access particularly to its content, i. e. some of the C++ source code of game entities, some game map sources and some model sources.

We have used Half Life 2 Multiplayer as a base game for our mod. Half Life is better documented by official Valve Developer Community [4] than other games and Half Life 2 Singleplayer version was not available with our version of SDK. C++ sources of our mod compiles into to dynamically linked libraries *Server.dll* and *Client.dll*. The game is executed as original multiplayer binary with these two dynamic libraries.

We have decided to incorporated SF-HRP ASM directly into *Server.dll*. Our ASM is connected to agent entity in game through proxy class. This solution minimises costs and latency of communication between agent and library as it is the most direct way of communication between engine and library with no unnecessary maintenance overhead. It also does not introduce any issues unrelated to subject of out thesis, as interprocess or network communication implementation issues.

Source Engine original implementation of AI of non-player characters (NPC) is based on schedules and NPC specific schedule selection method. Schedules consists of tasks and specify interrupting conditions. NPC's classes create complex,

[3]The "mod" is generally understood by community and used by Valve with respect to the Source Engine as a game modification

multiple inherited hierarchy with shared task, implemented in upper levels of hierarchy, common to all or only to specific NPC's (e. g. actors, player companions, enemy NPC's, etc.).

Task registration and schedule specification is done entirely with preprocessor macros, which is rather cryptic. It further complicates the situation that many AI-related problems are detected as late as in run time or even not detected and silently ignored (e. g. malformed schedules).

Tasks used in schedules also does not correspond observable actions on agent nor corresponds level of abstraction we presume in Chapter 1 . For example task sequence

- TASK_GET_PATH_TO_PLAYER
- TASK_RUN_PATH
- TASK_WAIT_FOR_MOVEMENT
- TASK_DOG_FACE_PLAYER
- TASK_FACE_IDEAL

corresponds action "Go to player". Situation is further complicated with model animations, which require dedicated task in order to play animation and any other necessary computations must be done in separate task. This actions are also poorly documented, which hardens extending NPC's capabilities beyond those already present in source code.

Conditions in terms of Source Engine are divided into non-explicit changes of internal properties of NPC's and explicit input functions and output properties. Input functions and output properties are exported via textual metafile describing their interfaces into Hammer Map Editor, where designer connects game entities through them.

In light of these facts we decided to represent agent's atomic actions in terms of SF-HRP as short, uninterruptible schedules in terms of Source Engine. Conditions are hard-coded on appropriate places in source code [4]. We present our sample wrapper as modification of original final NPC class in NPC hierarchy, thus inherited from base NPC classes and incorporated into hierarchy for better cooperation with engine. Textual action and condition related identifiers are translated into Source internally used enums by helper class.

Our agent - wrapper is based on a *Dog* entity which appeared in Half-Life 2 Singleplayer. Dog provided with SDK has only limited capabilities, i. e. movement, which is inherited from parent classes of entity, and capabilities regarding object catching and throwing. Adding new activities of the Dog demands changes in Dog's entity model and their propagation into source code which is beyond the scope of this thesis. However are Dog's limited capabilities problematic capabilities of the other NPC entities are similarly limited or limited only to the movement thus choice of another entity as base for the wrapper is not solving it.

[4] As their are already hard-coded by design.

2.7 Profiler

We provide only simple implementation of the profiler. More advanced implementation which collaborates with external tools is beyond the scope of the thesis. Our implementation displays information on execution of agent's behavior in *developers console* which is built-in textual output within source engine. Processed events are:

- activation of a condition
- deactivation of a condition
- selection of either action rule or explicit success rule or explicit fail rule

The instance of profiler is unique within library and with single global point of access. This implementation ensures that every object has access to the (same) instance of profiler without the burden of passing that instance around. We use *CInspectorHolder* class similar to *Singleton pattern* [15] for access to the profiler instance. The class is not intended for instantiation as Singleton, it provides static interface for event processing instead.

3. Scenarios

We presented SF-HRP ASM concept in Chapter 1 and implementation thereof in Chapter 2. In this Chapter we present scenarios and address issues related to the process of their creation.

The capabilities of the SF-HRP driven agent in our application are limited due to reasons described in Section 2.6. This decreases the complexity of possible scenarios thus presented examples are demonstrating only basic features. However this basic scenarios are sufficient to demonstrate certain bottlenecks.

The both of are scenarios are based on NPC called the *Dog*, object called *ball* and its manipulation through *physgun* - weapon capable of picking, catching and throwing of objects. All the items are pictured on Figure 3.1. Installation of the implementation and execution of the scenarios is described in Appendix C.

3.1 Scenario A

The scenario presents basic features of SRP. The Dog tries to grab the ball. When the Dog grabs the ball it throws the ball to the player. When the player gets the ball and throws it to the Dog, regardless if he caught it or picked it from ground, the Dog waits for catch. When the Dog catches the ball it throws it back. When the Dog miss the ball, it attempts to grab the ball from the ground and throw it to the player. The scenario then continues allowing to test different combination of player - Dog catches and misses of the ball. Source XML file of the scenario is included in Appendix B.1.

3.1.1 Conclusion

Given the conditions "dog has ball" and "ball is in the air" and actions for grabbing, catching and throwing the ball the design of Dog's behavior is rather straightforward as sees on included example. The only glitch is ever satisfied guard rule performing idle action.

Figure 3.1: In-game screenshot

Besides the straightforward design of the behavior we identify two issues in this Scenario. The Dog in scenario seem lazy, it takes observable delay before it goes for the missed ball though profiler displays both corresponding condition activation and rule selection simultaneously. After further inspection of the problem we have found the delay is caused by the way Source Engine manipulates the AI. According to the documentation [4] the engine calls methods resolving agent actions - lets the agent think - every one second. Originally used AI does not suffer this problem as it works on much lower version of abstraction. Actions we consider atomic are in this representation sequences of tasks 2.6 usually ended with suggestion on following sequence of actions.

The second issue is more technical and harder to observe. In some cases of manipulation with the ball and interaction with the player the Dog gets stuck idle though profiler is signalling proper condition states and proper action selection. After the further debugging of the source code we have found the Dog is unable to complete some task of which the Dog's action consists, e. g. thrown ball has collided with the Dog and interrupted the catching. In the original AI this problem is solved with task sequence interruptions on the lower level of abstraction than are the tasks. This is conceptual problem as the SF-HRP ASM does not address handling of low level events which is in Source Engine originally implemented as part of AI.

3.2 Scenario B

The scenario presents stateful features of SF-HRP ASM - the phases. In initialization phase the Dog grabs the ball. The execution phase is simple. If the Dog has the ball it attempts to throw the ball to the player and waits. When the ball hits the ground execution phase ends with failure. If player catches the ball the execution phase end with success. In termination phase the dog "dances" in case of success. In case of failure the Dog runs towards ball to see where the ball hit the ground. Source XML file of the scenario is included in Appendix B.2.

3.2.1 Conclusion

Given proper conditions and action similar to the condition and action in the previous example the design of Dog's behavior is still straightforward. The behavior is clearly divided into sections which we consider contributory. The complexity of execution phases is low in comparison to the Scenario A.

The problems are same as in previous example. Agent is particularly unresponsive and may get stuck occasionally.

Discussion

We believe the behavioural input format helps to decrease complexity of design of agent's behavior, because it reflects the SF-HRP concept with well-defined mapping of concept's primitives e. g. conditions, plans, rules, etc. to the XML elements.

The concept has proven simple from implementation point of view. The implementation of action selection mechanism is straightforward, the primitives of the concept are relatively separated and communication among them limited which simplifies the design. Used algorithms restricts to those described in the thesis and to the C++ Standard Template Library. Input file format allows parser to use libxml's SAX2 API resulting in lesser parsing time and smaller representation in memory than DOM API.

The library is in our implementation used synchronously. In every cycle are set elementary conditions and then requested appropriate action. For n being maximal count of elementary conditions in a condition and m number of elementary conditions changing its value is time complexity of conditions setting $\mathrm{O}(log(n) * m)$. Action selection uses STL maps for accessing primitives according to priority. With n being number of SF-HRP plans, m maximal count of HRP plans in an execution phase an o maximal count of rules in a HRP plan is due to fixed number of phases in SF-HRP plan resulting time complexity of action selection $\mathrm{O}(log(n) * log(m) * log(o))$.

Although the profiler was helpful with designing the agent's behavior more advanced implementation possibly communicating with external tools would be appropriate to evaluate the revenue.

The most disappointing has show the choice of virtual environment. Features of Source Engine with respect to AI reimplementation were unsatisfactory. Limits in available actions of the agent significantly restricts possible scenarios. Such scenarios does not test the SF-HRP concept in a manor we hoped for.

We have not found bottlenecks neither in implementation nor in timely fashion of the concept nor in use with testing scenarios. Although the lack of bottlenecks connected with scenarios is probably be caused by only very limited scenarios which are not testing SF-HRP capabilities properly.

Conclusion

We have designed input behavioural format. We provided implementation of SF-HRP ASM connected to 3D environment and example scenarios in the input format which can be tested with the implementation.

We have shown that the concept and the input format allow description of agent's behavior in simple, understandable way. State-Full extension of HRP concept has proven contributory in reducing HRP complexity.

However the choice of virtual environment has proven problematic. The engine limits available action of the agents and addition of the actions exceeds modifications of provided source codes. It restricts possible scenario to combinations of only a few actions. Another problem was that SF-HRP ASM concept is much more abstract than original AI mechanism which is closely related the engine implementation details.

The profiling of SF-HRP plan, even as simples as presented implementation is, has proven worthy both in design of the agent's plan and in addressing the engine issues.

Future work

We identify as the most significant task in future to overcome the limitations of chosen engine. Though we have shown contributions of SF-HRP ASM and input format with respect to the agent's design, believability of a behavior produced by the concept is still undetermined. It is uncertain whether the problem is related only to the chosen engine or is the feature of the concept. The implementation of the wrapper above *SF-HRP library* with different engine should give satisfactory answer.

There is also possibility of extending implementation of profiler for communication with external tools. Such an implementation would answer the question of full revenue of the profiler.

Presented implementation does not take all features of SF-HRP presented in [2] in account. Our library could be extended with some of these features, e. g. online modifiable be-trees.

Bibliography

[1] Brom, Cyril. Hierarchical reactive planning: Where is its limit?. In *Proceedings of MNAS - Modelling Natural Action Selection*. Edinburgh, Scotland, 2005.

[2] Plch, Tomáš. *Action Selectin for an Animat*. Prague, 2009. Diploma thesis. Charles university in Prague.

[3] Russel, Stuart; Norvig, Peter. *Artificial Intelligence: A Modern Approach*. 3e. Prentice Hall, 2010.

[4] *Valve Developer Community* [online]. 2011 [2011-08-03]. WWW: <`http://developer.valvesoftware.com`>

[5] *Unreal Development Kit* [online]. 2011 [2011-08-03]. WWW: <`http://www.udk.com/`>

[6] *The XML C parser and toolkit of Gnome* [online]. 2011 [2011-08-03]. WWW: <`http://xmlsoft.org/`>

[7] Bryson, Joanna J.; Stein, Lynn Andrea. Modularity and Design in Reactive Intelligence. In *Proceedings of the 17th international joint conference on Artificial intelligence*. San Francisco, CA, USA, 2001.

[8] *The Sims 3* [online]. 2011 [2011-08-03]. WWW: <`http://www.thesims3.com/`>

[9] Mateas, Michael. *Interactive Drama, Art and Artificial Intelligence*. Pittsburgh, PA, USA, 2002. Thesis. Carnegie Mellon University.

[10] Gemrot, Jakub; Kadlec, Rudolf; Bida, Michal; Burkert, Ondřej; Pibil, Radek, Havlicek, Jan; Zemcak, Lukáš; Simlovic, Juraj; Vansa, Radim; Stolba, Michaů; Plch, Tomáš; Brom, Cyril. Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents. In *Agents for Games and Simulations*. LNCS 5920, Springer, 2009.

[11] González, Pedro. P., Negrete, José; Barreiro, Ariel; Gershenson, Carlos. A Model for Combination of External and Internal Stimuli in the Action Selection of an Autonomous Agent. In *Proceedings of the Mexican International Conference on Artificial Intelligence: Advances in Artificial Intelligence*. London, UK, 2000.

[12] Friedlander, David; Franklin, Stan. LIDA and a Theory of Mind. In *Proceeding of the 2008 conference on Artificial General Intelligence 2008: Proceedings of the First AGI Conference*. Amsterdam, Netherlands, 2008.

[13] Pfeifer, Rolf; Scheier Christian. *Understanding Intelligence*. MIT Press, 2001.

[14] *Extensible Markup Language (XML) 1.0*. W3C, 2008. WWW: <`http://www.w3.org/TR/xml/`>.

[15] Shalloway, Alan; Trott, James. *Design Patterns Explained: A New Perspective on Object-Oriented Design*. 2e. Addison-Wesley, 2004.

List of Figures

List of Abbreviations

AI Artificial Intelligence

ASM Action Selection Mechanism

DTD Document Type Definition

HRP Hierarchical Reactive Planning, Section 1.1.2

SF-HRP State-Full Hierarchical Reactive Planning, Section 1.2

SRP Simple Reactive Planning, Section 1.1.1

XML Extensible Markup Language

A. XML input format

A.1 Document type declaration

```
<!DOCTYPE agent SYSTEM "sfhrp.dtd">
```

A.2 Document type definition

```
<!DOCTYPE agent [
<!ELEMENT agent (sfhrplan+)>
<!ELEMENT sfhrplan (cond, init, hrplan, term, cleanup)>
<!ELEMENT cond (elem|and|or|not)*>
<!ATTLIST cond id ID #IMPLIED>
<!ELEMENT elem (type, param*)>
<!ELEMENT and EMPTY>
<!ELEMENT or EMPTY>
<!ELEMENT not EMPTY>
<!ELEMENT type (#PCDATA)>
<!ELEMENT param (#PCDATA)>
<!ELEMENT init (list?)>
<!ELEMENT list (object+)>
<!ELEMENT object (#PCDATA)>
<!ELEMENT hrplan (plan+)>
<!ATTLIST hrplan entry IDREF #REQUIRED>
<!ELEMENT plan (rule+)>
<!ATTLIST plan id ID #REQUIRED>
<!ELEMENT rule (prio,weight,cond,flags,exec,sttout?)>
<!ATTLIST rule id ID #IMPLIED>
<!ELEMENT prio (#PCDATA)>
<!ELEMENT weight (#PCDATA)>
<!ELEMENT exec (#PCDATA|action)*>
<!ATTLIST exec type (success|fail|plan|action) #REQUIRED>
<!ELEMENT action (type|param+)>
<!ELEMENT flags (sticky|relsafe|intersafe)*>
<!ELEMENT sticky EMPTY>
<!ELEMENT relsafe EMPTY>
<!ELEMENT intersafe EMPTY>
<!ELEMENT sttout (#PCDATA)>
<!ELEMENT term (succ,fail)?>
<!ELEMENT succ (#PCDATA)>
<!ELEMENT fail (#PCDATA)>
<!ELEMENT cleanup EMPTY>
<!ATTLIST cleanup enabled (true|false) #REQUIRED>
```

B. Input examples

B.1 Scenario A

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE agent SYSTEM "sfhrp.dtd">
<agent>
  <sfhrplan>
    <cond id="do-catch-throw">
      <elem>
        <type>COND_DOG_CATCH_THROW</type>
      </elem>
    </cond>
    <init>
    </init>
    <hrplan entry="onlyone">
      <plan id="onlyone">
        <rule id="rule-catch">
          <prio>3</prio>
          <weight>1</weight>
          <cond id="dog-has-not-ball">
            <elem>
              <type>COND_DOG_HAS_BALL</type>
            </elem>
            <not />
            <elem>
              <type>COND_BALL_IN_AIR</type>
            </elem>
            <and />
          </cond>
          <flags></flags>
          <exec type="action">
            <action>
              <type>ACTION_DOG_CATCH_BALL</type>
            </action>
          </exec>
        </rule>
        <rule id="rule-get">
          <prio>2</prio>
          <weight>1</weight>
          <cond id="dog-has-not-ball">
            <elem>
              <type>COND_DOG_HAS_BALL</type>
            </elem>
            <not />
          </cond>
          <flags></flags>
          <exec type="action">
            <action>
              <type>ACTION_DOG_GET_BALL</type>
            </action>
          </exec>
        </rule>
        <rule id="rule-throw">
          <prio>1</prio>
          <weight>1</weight>
```

```
          <cond id="dog-has-ball">
            <elem>
              <type>COND_DOG_HAS_BALL</type>
            </elem>
          </cond>
          <flags></flags>
          <exec type="action">
            <action>
              <type>ACTION_DOG_THROW_BALL</type>
            </action>
          </exec>
        </rule>
        <rule id="rule-fallback">
          <prio>0</prio>
          <weight>1</weight>
          <cond id="true">
            <elem>
              <type>COND_TRUE</type>
            </elem>
          </cond>
          <flags></flags>
          <exec type="action">
            <action>
              <type>ACTION_DOG_IDLE</type>
            </action>
          </exec>
        </rule>
      </plan>
    </hrplan>
    <cleanup enabled="true" />
  </sfhrplan>
</agent>
```

B.2 Scenario B

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE agent SYSTEM "sfhrp.dtd">
<agent>
  <sfhrplan>
    <cond id="plan-state-full">
      <elem>
        <type>COND_DOG_CATCH_THROW</type>
      </elem>
    </cond>
    <init>
      <list>
        <object>ball</object>
      </list>
    </init>
    <hrplan entry="onlyone">
      <plan id="onlyone">
        <rule id="rule-throw">
          <prio>3</prio>
          <weight>1</weight>
          <cond id="dog-has-ball">
```

```
            <elem>
              <type>COND_DOG_HAS_BALL</type>
            </elem>
          </cond>
          <flags></flags>
          <exec type="action">
            <action>
              <type>ACTION_DOG_THROW_BALL</type>
            </action>
          </exec>
        </rule>
        <rule id="rule-succ">
          <prio>2</prio>
          <weight>1</weight>
          <cond id="player-has-ball">
            <elem>
              <type>COND_PLAYER_HAS_BALL</type>
            </elem>
          </cond>
          <flags></flags>
          <exec type="success" />
        </rule>
        <rule id="rule-fail">
          <prio>2</prio>
          <weight>1</weight>
          <cond id="ground-has-ball">
            <elem>
              <type>COND_BALL_ON_GROUND</type>
            </elem>
          </cond>
          <flags></flags>
          <exec type="fail" />
        </rule>
        <rule id="rule-guard">
          <prio>1</prio>
          <weight>1</weight>
          <cond id="ever-satisfied">
            <elem>
              <type>COND_TRUE</type>
            </elem>
          </cond>
          <flags></flags>
          <exec type="action">
            <action>
              <type>ACTION_DOG_IDLE</type>
            </action>
          </exec>
        </rule>
      </plan>
    </hrplan>
    <term>
      <succ>ACTION_DOG_REJOICE</succ>
      <fail>ACTION_DOG_SHAKE</fail>
    </term>
    <cleanup enabled="false" />
  </sfhrplan>
</agent>
```

C. Installation

The implementation requires licenses of Source Engine game as Source SDK is available only with these games. The installation process is:

1. install Steam client application, installation package and official support is available at

   ```
   http://store.steampowered.com/
   ```

   Do not be afraid nor to read the manual nor to contact the official support in any case of trouble.

2. run Steam client

3. install "Source SDK Base 2007" from tools section (select *Library* section, filter *TOOLS*, find "Source SDK Base 2007" in list of tools, right click and select "Instal Game..." from context menu)

4. copy directory "bcpoc" from included DVD into

   ```
   "C:\Program Files\Steam\SteamApps\sourcemods\"
   ```

   where

   ```
   "C:\Program Files\Steam\"
   ```

   is Steam installation directory

5. create batch file with content (or edit properly the batch file included on DVD)

   ```
   @"C:\Program Files\Steam\AteamApps\<account>\Source SDK
      Base 2007\hl2.exe" -dev -game "C:\Program Files\Steam\
      bcpoc" -allowdebug %1 %2 %3 %4 %5 %6 %7 %8 %9
   ```

   where

   ```
   "C:\Program Files\Steam\"
   ```

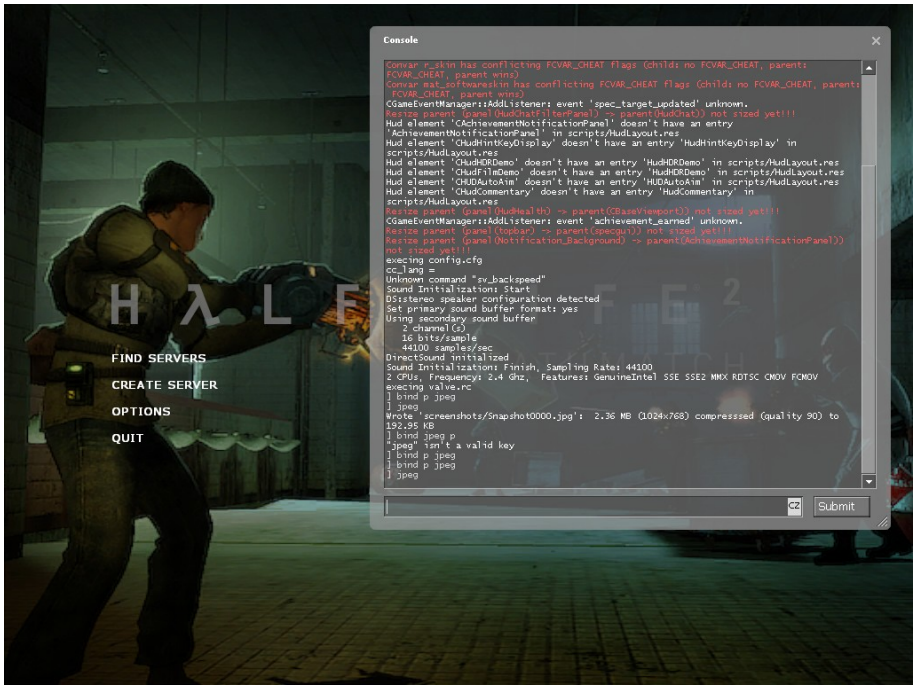
   is Steam installation directory and

   ```
   "<account>"
   ```

   is the name of used Steam account.

6. run the game via the batch file, in any case of trouble run "Source SDK Base 2007" either from Steam client or system tray Steam context menu. (system tray, right click Steam icon, select "Source SDK Base 2007") and try to run the game again

The scenarios are started via *Create server* → select *Map* → *Start* from main menu as seen on following picture. Each map corresponds a scenario.



bcpoc_ct - Scenario A

bcpoc_sf - Scenario B

Player holds *physgun* used for manipulation with the ball. The ball is pulled, grabbed and caught with right mouse button. The ball is thrown with right mouse button.