Utrecht University

Faculty of Science
Department of Information and Computing Sciences

Using Intelligent Agent for Automated Testing of Computer Games

Thesis

G.J. van Schie

Project Supervisor:
Dr. S.W.B. (Wishnu) Prasetya

Second Examiner:
Dr. F.J. (Fernando) Castor de Lima Filho

Abstract

Given the increasing complexity and the rising popularity of games, the importance of testing increases as well. Currently, a lot of the testing is done by alpha testers because of their flexibility in handling changes in level designs and their ability to find bugs in a lot of different types of games. Using automation to perform part of this testing could reduce development costs and increase the speed at which bugs are discovered. Agent-based testing approaches offer flexility to deal quickly with changes in the environment and the possibility to define general strategies to reach a goal in a random environment.

During this research, the iv4XR framework is used to implement an agent-based tester to automatically play and test NetHack, a procedurally generated game. The effectiveness of this agent is assessed using code coverage, mutation testing, and monitoring of properties defined by LTL formulas.

The agent achieves low code coverage (14.2%) and a mutation score of (56.8%). Monitoring properties using LTL does show invalid state transitions within the game can be detected automatically. Improvements to the iv4XR framework are found for future use of the framework. Some of these improvements have been implemented.

This research concludes effective play testing using the agent has not been achieved. To increase effectivity of the agent, an improved strategy must be defined. Using LTL formulas to test properties in the game could greatly assist alpha testers by automatically monitoring a play through. Their potential will need to be investigated in future research.

Keywords — Automated game testing, Agent-based testing, Iv4XR framework, Aplib, Code coverage analysis, Mutation testing, LTL monitoring

Table of Contents

1 Introduction 3

2 Preliminary 5
2.1 NetHack 5
2.2 Agent-based testing 6
2.3 The iv4XR framework 7
2.3.1 Goal/Tactics/Actions/Predicates 7
2.3.2 LTL component 7

3 Research Questions 9
3.1 RQ 1: what testing effectiveness can be achieved? 9
3.2 RQ 2: can LTL be used to facilitate testing? 10
3.3 RQ 3: how can iv4XR be improved for testing? 10

4 Methodology 11
4.1 Intelligent agent iv4XR 11
4.1.1 Agent concept example 11
4.1.2 Translation of concept to code 12
4.2 Architecture 13
4.3 NetHack state 14
4.3.1 Map information 14
4.3.2 Monster information 14
4.3.3 Entity information 14
4.3.4 Static information 14
4.4 Socket connection 15
4.5 iv4XR state 15
4.5.1 iv4XR WorldModel 15
4.5.2 iv4XR environment 15
4.6 Agent functionality 15
4.6.1 Navigation 16
4.6.2 Item interaction 16
4.6.3 Object selectors 16
4.7 Agent strategy 16
4.7.1 Survival 17
Nutrition 17
Handling monsters 17
4.7.2 Interaction 17
4.7.3 Exploration 17
4.8 Verification using LTL formulas 18
4.9 Coverage 18
4.10 Mutation testing 19
4.10.1 Repeatability 19

5 Implementation 20
5.1 StepState and GameState 20
5.2 Navigation improvements 20
5.2.1 Existing implementation example 21
5.2.2 Improved implementation 21
5.3 WorldModel 22
5.3.1 Existing WorldModel 22
5.3.2 Improved WorldModel 23
5.3.3 Existing WorldEntity history 23
5.3.4 Improved WorldEntity history 23

6 Results 24

6.1 RQ 1: testing effectiveness results 24
6.1.1 RQ 1a: code coverage results 24
Coverage baseline 24
Coverage comparison 24
Line coverage iv4XR 25
Line coverage BotHack 26
Reflection on RQ 1a 27
6.1.2 RQ 1b: mutation results 27
Generated mutations 27
Camera entity 27
Reflection on RQ 1b 29
6.1.3 Reflection on RQ 1 29
6.2 RQ 2: facilitating testing using LTL 29
6.2.1 LTL properties results 30
6.2.2 Reflection on RQ 2 30
6.3 RQ 3: what improvements can be made to iv4XR? 30
6.3.1 Conditionals 30
Add goal structure 30
Add multiple 'on' predicates for actions 31
Add a condition structure for a tactic 31
6.3.2 Logging actions 31
6.3.3 LTL clarity 31
6.3.4 Reflection on RQ 3 32
6.4 Reflection on the main RQ 32

7 Related Work 33
7.1 Automated game testing approaches 33
7.1.1 Random agent testing 33
7.1.2 Model-based testing 33
7.1.3 Search-based testing 34
7.1.4 Machine learning 34
7.1.5 Agent-based testing 34
7.2 Game testing verification 35
7.2.1 Coverage testing 35
7.2.2 Number of bugs found 35
7.2.3 Mutation score 36
7.3 Testing using LTL 36

8 Conclusion & Discussion 37
8.1 NetHack as GUT 37
8.2 Assessment of the agent 38
8.3 Iv4XR framework reflection 38

9 Future work 39
9.1 Improve agent strategy 39
9.1.1 Improve current agent strategy 39
9.1.2 Using goal structures 39
9.2 Explore the usability of testcases preparation 40
9.3 Assist testing using LTL properties 40
9.4 Simulating personality to improve testing effectiveness 40

A Mutation testing results 44

B Nutrition tactic 46

Chapter 1

Introduction

Increase in game size, expected quality, and complexity increases the required time spent on testing[18]. Because testing games is very difficult and changes often happen, testing games is often still performed manually by humans, commonly referred to as alpha testers. These people take a lot of time and effort to verify game logic[30]. Alpha testers try to find bugs in the game and report them back to the developers. Because alpha testers are expensive, it is not always possible to use alpha testers. So it can take some time between developing the code and the testing phase. The goal of this thesis is to explore ways to cut expenses by automatically detecting as many defects as possible before utilizing alpha testers. Because many games are very complex, some of the other researches are not an accurate representation of how good their testing implementations are for complex games.
NetHack is the complex game selected for this thesis. It is a very complex game with a lot of interactions, randomly generated maps, and difficult enemies. A challenge to create a bot that automatically can play the game was held, however machine learning proved to be less effective than programmatic implementations using tactics and strategies to play the game[2]. We will compare our automated playing approach with a bot implementation that has been developed over years and can give insights in what is achievable in NetHack. During this thesis, there is a time constraint, so the strategy of the agent-based implementation is not as sophisticated.
Agent-based testing is not the only testing implementation. For example, QuickCheck[12] and EvoSuite[16] are popular testing methods to automatically generate random inputs, so the test suite will test properties rather than individual values. This method of testing makes it generally more robust than unit testing. Such a testing approach is well fit for a typical program. However, for a complex game like NetHack the state of the game is not possible to generate. Agent-based testing is used because we specifically want to test the iv4XR framework. This is an agent-based testing framework, which observes the state and creates a belief state from all observations. The agent will have a goal, to reach it tactics with actions will be executed. Each turn it will re-evaluate which action it should do next, so if during exploration a monster appears it will first deal with that before resuming the exploration.
Since gameplay changes during development, it is relevant to test using an approach capable of dealing with changes within the game and levels. This is another reason to choose for a robust testing method like agent-based testing. The goal of this thesis is to research whether it was possible to test NetHack using an intelligent agent-based approach. In other research projects, when intelligent agent-based testing is used, it is performed on a simple game with specially introduced bugs or machine learning is used on a real game[29, 41]. Using machine learning for automated play testing and finding a wide variety of bugs has become very popular in the recent research. During this thesis, we will reflect upon why NetHack is not a good candidate to automate using machine learning approaches.
This thesis is a continuation of two pieces of research. One was performed by a previous student, who used the iv4XR framework on a simpler clone version of NetHack called NetHackClone[29]. And a more recent research thesis discussing the performance of the iv4XR framework on three example games and a note on the experience whilst using the framework[43]. One of the games mentioned there is similar to NetHack, called MiniDungeon, the latter research also optimized part of the agent implementation written by the student.
The gap this research tries to fill is introducing mutation and LTL property testing in the game testing industry. Currently, mutation testing and LTL property testing are almost not found in existing researches, even though mutation testing is very widely adopted in other parts of software testing. And LTL property testing can test properties on the play-through level, which is the same level as manual testers. This research aims to show these methods are also suitable for testing games.

Contribution

Work done to perform this research, other tools and bot implementations were tested, but only the selected tool is shown in the final work. The end result contains an adapted NetHack and NLE code to share more about the internal state of NetHack. Also, in Java configurations it is possible to configure multiple settings, for example using seeds, selecting the player type, and replaying a previous play through. The socket implementation and API to have fast communication between the socket server and client. Code to interpret the information on the client side and the object model in which the game state is kept and updated after each game step. Regarding the agent framework, some improvements have been made to the already existing iv4XR framework. Because the framework provides an interface to create an agent, creating the instance of the agent state, environment, and goal with strategies was done for NetHack specifically. Lastly, scripts were written to easily install, build, and run the different components.

Thesis structure

The rest of the thesis will be structured as follows. First, some theory will be given to understand NetHack and some concepts (chapter 2). Then the main research question and the sub research questions will be described to understand what the thesis is trying to answer (chapter 3). Then the methodology and the implementation will be discussed in detail to describe the selected tools for the tests and some improvements made to the iv4XR framework (chapter 4 and chapter 5). Afterwards, the results of the research will be shown in addition to the answers to the research questions (chapter 6). The research of this thesis will then be compared to other found research (chapter 7). And then finally a conclusion will be given with recommendations for future work (chapter 8 and chapter 9). Finally, there will be a short appendix at the end with some test results.

Chapter 2

Preliminary

The goal of this chapter is to provide some general information about the complex game NetHack that is being tested. Additionally, information will be given about agent-based testing on a more theoretical level to understand the origin and general structure. Finally, the iv4XR framework is discussed, with some features of the framework relevant to the research. This includes the LTL component of the framework.

2.1 NetHack

The game under test (GUT) chosen for this thesis is NetHack[32]. This is an old terminal game dungeon crawler and uses ASCII characters to render the map for the player. An example of the game overview is shown in Figure 2.1 with explanation about what information is shown on the GUI.

Figure 2.1: *An example NetHack GUI overview using the ASCII characters[37]. At the top of the window, messages are displayed with player observations. The middle area is reserved for the current level. Corridors are the squares with the '#'. Floor tiles are indicated by '.' and are surrounded by walls '|' or '-' and doors in brown using '+', '-', or '|' characters. The highlighted '@' is the player with the green '@' being another human. Letters are monsters, for example, the blue 'G'. Many of the colourful ASCII characters are items the player can interact with. At the bottom there is an overview of how skilled a player is in different aspects followed by the score of the player, which increases by defeating monsters. Lastly, there are indications of the depth of the player in the dungeon (Dlvl), how much money the player has ($) the health and other information like how much hunger the player has.*

Each game is different because the dungeon, items, and enemies are randomly generated. The goal of the game for the player is to collect a gem at the bottom of the dungeon and bring it back to the start of the dungeon. Meanwhile,

the player must defeat enemies and collect items such as food and weapons to survive. There are different types of characters to select from at the start of the game, each having their own skills and abilities. It can take 100,000 turns to reach this goal of the game. There are more than 100 different commands possible to interact with items or navigate through the dungeon in order to reach that goal. Not all of these commands are required for reaching the goal, however it helps demonstrate the complexity of the game. NetHack has been selected because it is a complex game with randomly generated maps, items, monsters, and many interactions. The complexity and randomness make automated testing especially difficult and manual testing time-consuming. A robust approach is required to perform tests, even when the levels change.
Properties that make this game good for testing are the turn-based progression and the grid-based navigation. The turn-based gameplay allows the testing method to be uninfluenced by the speed of the testing solution. The grid-based map in NetHack makes navigation through the levels easier, since there are only a limited number of locations the player can be at. Another reason why this game is selected is the OpenAI Gym[10] interface, created in the NetHack Learning Environment (NLE)[27], which can be utilized. The development of NLE was to hold a challenge[1] for AI researchers to create learning algorithms and test their effectiveness in playing NetHack. NLE makes the translation to a game state significantly easier. And, in turn, allows us to spend more time on programming and conducting research.

2.2 Agent-based testing

Agent-based testing is an automated form of testing based on the belief-desire-intention software model (BDI) originating from philosophy[9]. This model has been adopted in efforts to create computer agents with human-like reasoning. An overview of the BDI cycle is shown in Figure 2.2.

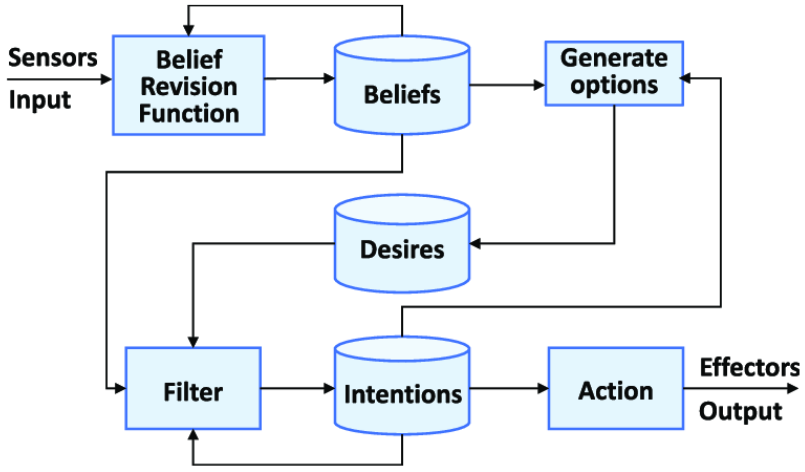


Figure 2.2: *An overview of the BDI cycle[45]. The real state of the world, referenced to as input. An observation of this world is done by the revision function. The beliefs of the agent are updated by observations and this is the observation cycle, these beliefs then form a belief state. Based on beliefs of the agent, a new desire can be created. Goals, referred to as desires in the figure, and the belief state are used to decide what intention the agent has. For example, if the agent has the desire to get to the next level, it is only possible to act on this goal when the agent believes there to be a portal to the next level. This is the purpose of the filter function. One of the desires then is selected, through intentions the desire can be fulfilled or updated. Performing an action can help reach the goal, like moving towards the portal for the next level. Or the desire can be updated or removed when the agent knows there doesn't exist such a portal in the current belief state. Only actions update the external state the agent interacts with. When that happens, the belief state is updated with a new observation and again an intention is selected using the filter function.*

Agent-based testing can act upon changes quickly because after each action, a new observation that updates the belief state is performed. So, agent solutions are used in environments that can be perceived and acted within, sometimes autonomously. And as a result of that change a with the filter function, the agent can suddenly focus on a new desire or have a different intention. Agents could have several functionalities like autonomous operation, in-

teractions between multiple agents, reacting to changes in the observed environment, and goal oriented behaviour with sometimes improving the strategies to improve decision taking in future[26].
When applying agent-based testing to a game, the environment which is observed can be the direct internal state of the game or a less complete observation. If the game is written in the same programming language as the testing framework, it is easier to have access to the entire game state. However, the agent only needs the information to reach the goal and be able to observe situations it needs to react to. This means not all the information of the game needs to be provided for the agent to interact with the game.
To perform interactions within the game, the player often needs to be at or around the location of an item or part of the map. This is why pathfinding is essential for a testing agent. It is a great challenge to perform this pathfinding since during navigation the path can change when obstacles in the map can be dynamic. Other items or creatures can block the path to the objective. In NetHack the agent can only move over a grid, so it is possible to perform a pathfinding algorithm each frame of the game without long computation time. The implementation of navigation will not use sophisticated logic to deal with the dynamic obstacle problems. If at a certain time step, the agent cannot continue its desired path, it will wait until the obstacle is removed.

2.3 The iv4XR framework

This framework iv4XR, also referred to as Aplib[42], was created by the iv4XR project team to test games using the agent based testing model. What makes this framework novel is the imperative programming control, making it more intuitive to program agents with it.
These intelligent agents provided by the framework allow for testing games in a robust way by not having to make assumptions about the game before starting a test. By writing broad testing strategies, changes in the level layout, item locations, and monster locations will not prevent the agent from progressing within the game.
The framework is written in Java and provides some useful interfaces to perform game testing using an BDI approach. The foundations are programmed to let an agent explore the world it is connected to. Also, the agent observation cycle can easily be used, and the world model state can be queried. These interfaces are created during earlier experiments and research projects using the framework.

2.3.1 Goal/Tactics/Actions/Predicates

If there are goals which include long term planning, and it can be split into sub-goals, it is possible to do this by creating multiple goals and letting tactics add new goals for the agent. There can be an ordering in the goals, deciding which goal is currently being worked on. This feature is not used in the current research, so this will not be discussed in detail, the agent has one general goal in all examples. Goals provide logic to indicate whether the goal has been reached or has been aborted. It can be aborted when the agent has died, for example.
An agent aims to complete its goal, it will continuously execute the pre-defined tactic which are meant to help the agent achieve the goal. A goal can for example be achieved by performing several actions in sequence. Because in NetHack an action can change the environment and cause the player to react, a more reactive tactic to achieve the goal is used. The 'FIRSTof' tactic is used, which performs the first 'enabled' action in the current state. This process is repeated each step until the goal is completed. If there are no actions enabled, but the goal has not yet been completed, the agent can abort the run or wait until an action is available again.
Any action can be guarded to disable it. If an action attacks an enemy if it is adjacent to the agent, a guard will be written for it, preventing the action from being executed without an adjacent enemy. This guard, also called predicate, can be a boolean or object, the predicate is enabled if true or in case of the object when it is not a null object.

2.3.2 LTL component

Another feature in the iv4XR project is a linear temporal logic (LTL) component. LTL is a language to evaluate a condition over finite or infinite executions. The logic language syntax contains time related grammar in order to specify the order of states and boolean operators to specify which properties should hold. The LTL grammar has propositional variables which are the input variables, these can be either true or false in a state. The language also contains logical operators to combine the boolean variables. The final part of the language are the temporal modal operators, which are shown in Table 2.1.

Unary operators		
X	Next	P holds the next state
F	Finally	Eventually P holds
G	Globally	P always holds
Binary operators		
U	Until	P holds until Q, Q must hold eventually
R	Release	P holds until P and Q hold, then neither is required to hold
W	Weak until	P holds until Q, Q does not need to hold
M	Strong release	Variation of release where P and Q must hold along the execution

Table 2.1: *Temporal modal operators in the LTL language. P and Q are propositional variables. Unary operators only use one propositional variable, whilst the binary operators require two propositional variables.*

The LTL language has many operators and logic expressions. Since the language validates properties over a linear line of states as shown in Formula 2.1 it can check properties over executions during runtime. After the execution, the result of whether the properties hold can be examined[17]. Games run at discrete steps with update loops, this creates a discrete chain of states over which an LTL property can be verified. An LTL formula that holds globally over the entire execution is also referred to as an invariant.

$$S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_{...} \rightarrow S_n \tag{2.1}$$

Formula 2.1: *A discrete chain of states LTL can verify, also called an execution.*

LTL properties allow for monitoring the global state[48]. An example is a global property to monitor the health is a positive value and not greater than the maximum health. Another example is monitoring all the positions of the player and monsters within the game are within the boundaries of the level. A limitation of LTL is only the current node is accessible. So it is not possible to monitor incorrect state changes. To solve this, each state S has a reference to the previous state S'. With that addition, it is possible for example to monitor a score does not decrease during the run as shown in Formula 2.2.

$$ScoreNotDecreasing : G(Score(S') <= Score(S)) \tag{2.2}$$

Formula 2.2: *Example LTL formula, using the previous state to monitor a non-decreasing game score. The initial game state will be handled separately.*

LTL in a finite execution has a different definition for some of the operators[6]. If a formula containing the next operator is triggered in state S_i, there must exist a state S_{i+1} after that to check the formula. If state S_{i+1} does not exist, the formula will be unsatisfied.

Chapter 3

Research Questions

This chapter aims at specifying the research questions the thesis will answer and explain why the sub research questions are important and help answer the main research question. The main research question is whether agent based testing, using the iv4XR testing framework, allows for testing of complex games. The sub research questions used to answer the main research question are:

1. What testing effectiveness can be achieved?
2. Can LTL be used to facilitate testing?
3. How can iv4XR be improved for testing?

3.1 RQ 1: what testing effectiveness can be achieved?

To answer this research question, two verification methods have been selected to assess the testing effectiveness for NetHack. Because of this, the research question will be split up into two, namely RQ 1a and 1b. Each will assess the effectiveness separately, and the observations will later be evaluated to give an indication of how effective the testing is.

RQ 1a: *what code coverage can be achieved?*

Code coverage is a measure of how much of the program code has been executed. It follows the program path and can show afterwards which lines of the code were run during execution. If a line is not executed during the play-through, it is also not able to find mistakes in the code. Using code coverage tools, the coverage of the agent testing implementation can be inspected. The coverage is to be collected over multiple runs, and averaged to get accurate results to measure the coverage without being heavily influenced by outlier results. This measure is also used in other game testing research, so that also why it is included.
Unfortunately, there are no existing tests available for NetHack to compare coverage results with. A bot implementation that can automatically play through the game will be used as a baseline to compare against. A bot does not aim at achieving high code coverage, but does give an indication how much coverage a play-through is able to achieve.
A comparison of these two coverage numbers lets us see whether this research will reach a coverage percentage that can achieve effective play testing.

RQ 1b: *what mutation score can be achieved?*

Mutation testing is automatically changing, called mutating, part of the code, execution log, input values, or other parts. Mutating part of the code can be replacing a boolean operator or constant in the code by another operator or value. The resulting mutation score represents what fraction of mutations is detected during testing, detection being a failed test suite. So it gives an indication of how robust the part of the code is tested. If the robustness of the test suite is higher, then it is more effective at finding bugs when they are accidentally introduced in the code.
There are several layers at which mutations can take place. In answering this research question, we will look at mutations at the source code level. These mutations do not require knowledge about how the game works and does give a representation of bugs that can occur when developing code. The bugs that will be introduced are tiny

adaptations to the source code. These bugs are most of the time found using unit testing instead of play-through testing.
The reason for not introducing play-through bugs, for example not executing an action, is because these bugs are artificially selected. Choosing an automated mutation tool will avoid a bias in choosing which bugs are introduced. Even though, mutation testing is not commonly used for game testing. Because it is an automated way of introducing bugs, it can be a fast method to inspect how effective any created testing framework is.

3.2 RQ 2: can LTL be used to facilitate testing?

In game testing, LTL properties is rarely used for testing. The standard method is using pre- and post-condition. Testing using LTL properties allows for testing time and order dependent properties in a linear state sequence. This corresponds with discrete steps of a game execution. This is why LTL as a method to test is explored. The properties that can be defined over a global state and the state changes also correspond more with the level at which alpha testers will verify the game. Rather than exact pre- and post-condition checks in code, verified by unit tests.
We will investigate whether automated testing can be improved by the use of LTL formulas. These formulas can be automatically checked over a play-through. This play-through does not need to be performed by an automated testing solution, and instead can verify any run. If there is a violation of a property, it potentially indicates the presence of a bug in the play-through. After this faulty game execution is found, it can be saved and replayed. And if it was indeed a bug, the game can be updated and the same steps can be performed to use as a testcase. The fix to the bug can then be automatically tested as a check before performing testing in future.
Another benefit of LTL formulas is the possibility to add them to the GUT without adapting the original with assertions or additional methods in the original source code. This is also why it is worth exploring using LTL for helping with testing.

3.3 RQ 3: how can iv4XR be improved for testing?

Besides researching the capabilities of the agent-based framework, another goal is to investigate how the framework can be adapted or extended to improve the testing experience for future research. The aim is to make use of different features of the testing framework to propose improvements to several aspects.
Initial experience with the framework indicated that it was difficult to prevent duplication. For example, having multiple identical tactics when using multiple goals. In many cases, the goal needs a section in the tactic aimed at surviving. Surviving can be by attacking enemies that pose danger or waiting for health regeneration during moments with low hp. Adding another goal also needs to contain the tactic for surviving, since survival is important for any tactic.
During this research, we will look for a solution to this problem or see whether there are other parts of the iv4XR framework that can be improved. If possible or required, improvements can be made during the research as well.

Chapter 4

Methodology

This chapter will explain how an intelligent agent in iv4XR works and how through simple tactics and actions more complex behaviour can be defined. Then we will go in depth about the architecture of the different components of the program and discuss what information the agent state has. Finally, some tools used during the thesis will be mentioned and argumentations will be provided for their choice.

4.1 Intelligent agent iv4XR

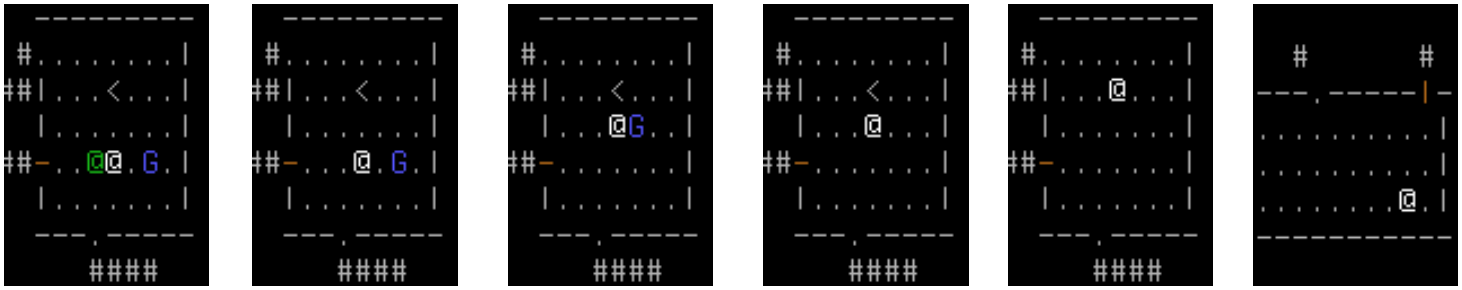
To understand how the agent works, first a small concept of an agent goal and tactic will be demonstrated. A translation of that tactic to code will be presented. Finally, the general structure of a goal will be discussed shortly.

4.1.1 Agent concept example

In the left image of Figure 4.1 the agent is in a room with two enemies. The goal of the agent is to reach the next level. However, if the agent would ignore the enemies, the agent could get killed and fail the goal. Because of this, dealing with the enemies will be given a higher priority. The order of the actions is used to enforce this priority. A tactic can be for example be 'ANYof', selecting an action at random, 'SEQ' performing each action once in order, or 'FIRSTof' each iteration selecting the first enabled action. In this example, we use the 'FIRSTof' operator for the tactic. The actions in the tactic are shown below.

1. Attack adjacent enemy
2. Navigate to stairs
3. Move downstairs

Each iteration it will first check if there is an adjacent enemy, if that is the case it will perform the first action, attacking the enemy. If there is no adjacent enemy, it will navigate to the stairs using pathfinding. Finally, to finish the goal it will move downstairs, when the agent has completed navigating to the stairs this action will be performed. An example of an execution is shown in Figure 4.1.



(1) Attack enemy left (2) Navigate up (3) Attack enemy right (4) Navigate up (5) Move downstairs (6) Goal completed

Figure 4.1: *The player (white @) is in a room with two enemies (green @ and blue G). The agent has the intention to move to the stairs (<) and move downstairs. Under each step of the sequence, the action of the player is given. The pathfinding to the stairs is in a straight line. By performing this tactic, the agent is able to complete the goal.*

As shown in the example, a short tactic can create an agent capable of achieving a goal within a game. It is clear this tactic is a robust method to achieve the goal, it does not matter if the enemies or the player are in another location in the room. This general definition can handle much more complicated scenarios as well and respond quickly when the situation changes in the game. Making the tactic more elaborate can make the agent behaviour seem intelligent.

4.1.2 Translation of concept to code

The translation of the concept, using the iv4XR framework, is shown in Listing 4.1. It also contains how the goal structure is added to the agent and how to initiate the agent loop.

```
1  public static void runAgent() {
2      GoalStructure G =
3        goal("Reach next level")
4            .toSolve(
5                (Pair<AgentState, WorldModel<Player, Entity>> environment) -> {
6                  NetHack app = environment.fst.app();
7                  return app.gameState.stats.dungeonLevel > app.previousGameState.stats.
                       dungeonLevel;
8                })
9            .withTactic(
10               FIRSTof(
11                   // 1. Attack adjacent enemy
12                   Actions.attack()
13                       .on(
14                           (AgentState S) -> {
15                             Monster monster =
16                                 MonsterSelector.adjacent.apply(S.app().level().monsters, S);
17                             // No adjacent monster found
18                             if (monster == null) {
19                               return null;
20                             }
21                             // Provides a direction to the attack command
22                             return NavUtils.toDirection(S, monster.loc);
23                           })
24                       .lift(),
25
26                   // 2. Navigate to stairs
27                   NavTactic.navigateToTile(TileSelector.stairDown),
28
29                   // 3. Move downstairs
30                   Actions.singleCommand(new Command(CommandEnum.MISC_DOWN))
31                       .on_(Predicates.on_stairs_down)
32                       .lift(),
33
34                   // 4. Otherwise abort
35                   ABORT())).lift();
36
37     // Creates agent and start the loop
38     TestAgent agent = new TestAgent(Player.ID, "player").attachState(new AgentState()).
           attachEnvironment(new AgentEnv(new NetHack())).setGoal(G);
39
40     agent.runloop();
41 }
```

Listing 4.1: *A translation of the concept tactic is shown. To check whether the goal has been completed is defined from line 4-8. When the current dungeon level is greater than the previous level, the goal is completed and the agent will stop. The first action aimed at attacking adjacent monsters is the most extensive action. The action defines an attack, however because of the predicate in the 'on' function it will only attack if there is an adjacent monster. The attack will be aimed at the direction of the adjacent monster. The second action uses the pathfinding to get to the stairs. When the player is on the stairs down, it will use the command to move down. An abort in added, so the agent stops if there is no enabled action. Otherwise, the agent will continue even if the level does not contain stairs. Finally, an agent instance is created with state and environment and the loop is started.*

The goal structure definition has been shown. The goal contains a name, when the goal is completed iv4XR uses this name to show which goal is completed. The game state is used to check whether the goal of reaching the next level is completed. The goal has an attached tactic to complete this predicate. The three actions can clearly be seen in the tactic definition. The 'on' function has a return value in case the guard contains useful information for the action. in this case, the guard knows in which direction the adjacent monster is and will provide this to the attack action. If a null object is returned, this action is disabled and the tactic will see if it can perform the next action in the tactic. A special class is used to query the game state for tiles and other objects in the game. So the navigation to stairs only needs to query object. Finally, when moving downstairs, the agent will check whether it is currently on a stairs tile that goes down. If this is the case, the agent will perform the command to go down, then the goal will be completed. If there would be no stair that goes downstairs in the level, the 'ABORT' action is triggered to prevent the agent to continue the goal structure without updating the game state by performing a command. Lifting the goal is done to make a goal structure from it. Goals can be put in a structure to repeat a sequence of goals continuously, but this functionality is not used in this research.
The code to run the agent is also included to show how to attach this goal to an agent and start the agent. Some details to have more control over when the agent stops is emitted from the example, but the code gives an accurate description of what it takes to create an agent and let it automatically play the game. This tactic is of course larger if the agent has to deal with more complicated examples and contains exploration. The architecture of the research program will now be explained.

4.2 Architecture

First, the overall setup of the program will be explained. This is to make it easier to understand when components of the program are discussed. These different components work together in order to run NetHack. The schema is visible in Figure 4.2. The reason for a representation of NetHack in Java is because iv4XR needs the state of the game to choose which action to perform. Because NetHack is written in another programming language, it is not possible to have direct access to the internal state. An object representing the NetHack game state is created and updated with each observation.

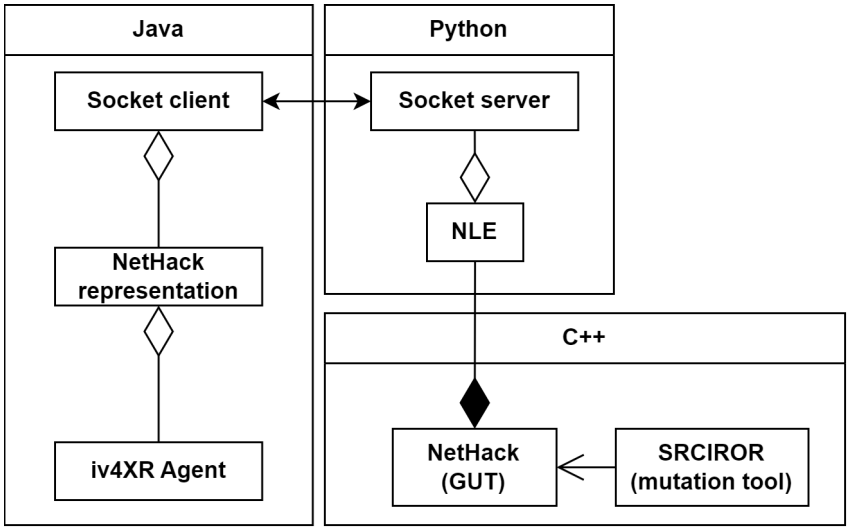


Figure 4.2: *At the right bottom corner we have the GUT NetHack which contains all the logic. NetHack is part of the Python package called NLE. Via a socket server and client connection, the data is transferred using byte messages. This information is then interpreted and a game state representation of NetHack in Java is created. The iv4XR agent can use this game state to perform the BDI cycle on and select which action to perform. This action then will be sent from Java to NLE and NetHack will get updated. It is also possible to perform mutations on NetHack using SRCIROR. This can change the NetHack source code and a test can be performed to check whether the mutation is detected.*

4.3 NetHack state

The NetHack version was adapted for the NetHack Challenge to share parts of the internal state with NLE in Python. This has been extended during this thesis to share more information about the internal state with NLE. The Gym environment in NLE is a collection of arrays representing information about the map. By default, it only exposed information a player of the game has. A testing solution can require more information about the internal state to assert properties, or in this case help with automated testing. Some of the changes will now be discussed.

4.3.1 Map information

Originally, only information about what is visible on the map is available in NLE. For example, hidden corridors and doors in the map are not visible for the agent. These can be found through searching. However, searching walls early in the game has a low probability to find these parts. It is difficult to determine how long the agent needs to search a given part of the level because searching is non-deterministic. It can take many search actions before the hidden section will be revealed. If the agent does not know where hidden parts of the map are, it can cost a lot of turns to find these sections, and possibly cause the player to run out of food and die.
More map information is shared from NetHack to NLE. This includes information about each position tile and state, for example whether the door is open or closed, and whether it is hidden on the map. All this information is known to the agent and shared when it enters a level.

4.3.2 Monster information

The shared information about the monsters was limited in a few aspects. Only the visible monsters would be shown, this could mean when the agent was blind it would not be aware of any monsters. By default, it was not known if a monster was close to the player. Also, because monsters move around the map, it was not possible to reliably track a monster over multiple turns. Because when a monster would disappear, it was not known whether the monster had died or moved out of the player vision. Also, it was not possible to know whether a monster was hostile towards the player or passive. When the agent attacks a passive monster like the shopkeeper, the agent immediately died and the run would be over.
Information about the monster position, ID, hostility state, and their type was shared to keep accurate track of all the monsters in the level and avoid attacking passive enemies. Some attacking tactics have been specified differently for certain types of monsters.

4.3.3 Entity information

Similar to the information shared about monsters, the information that was shared by NLE about entities was limited. Information shared about an entity would be its type and a semi-random entity identification number called glyph. If two of the same potion types would be present in the level, the glyph would be the same. Some aspects like potion effects are however randomized. So the effect of a potion with glyph 5010 could heal one run, but the next run it would make the player sick.
ID information is also added for the entities to keep track of unique entities. The randomness of glyphs was also removed to know at the start of a run the exact entity. Lastly, the position, quantity, and turn the entity was created were included in the shared state.

4.3.4 Static information

The additionally shared information about the internal state discussed so far is dynamic, for example position of monsters, and needs to get sent every update of the game. This part is about static information of entities and monsters that do not depend on the current game state. The name and weight of an entity are static. This information is gathered from NLE and saved in JSON files, when an entity in the game state is observed the static information is added automatically to the object together with the dynamic information. Information that is being stored for monsters and entities are seen below.

Static information of monsters

- Name
- Monster type
- Minimum level
- Movement speed
- Resistances
- Weight
- Nutrition
- Difficulty

Static information of entities

- Name
- Entity class
- Weight
- Cost
- Weapon skill
- Is missile, and does it get shot from a weapon

The name of a monster is used for example to only attack the monster with ranged weapons. The information about weapons skill provides information about whether it is a weapon and whether it is ranged. If it is ranged, additional information is given, so the agent knows whether it has ammunition to use the weapon.

4.4 Socket connection

The information about the map, monsters, and entities are shared every update of NetHack because it is dynamic information of the game state. From the others side the player input, dungeon seed, player type has to be communicated with NLE. This two-sided communication between Python and Java is facilitated by a socket connection. To keep the overhead of message parsing low, communication is strictly binary. This reduces the size of the stream that is sent each step significantly. Using a socket connection also allows the socket client and socket server to run on different devices.

4.5 iv4XR state

An iv4XR agent implementation contains 3 different parts. These three components will be discussed now to get understanding of how an agent is created in iv4XR. The iv4XR agent state contains the iv4XR environment, navigation object, and WorldModel. It is responsible for updating the WorldModel, navigation, and contains some functions to retrieve entities from the WorldModel. The other two components will be explained below.

4.5.1 iv4XR WorldModel

The WorldModel keeps track of entities. Each observation, the information of the player and each entity in the world is observed, put in an WorldEntity object and added to the WorldModel. Because there is an observation cycle with a new WorldModel each time, the WorldModel provides an automated method to merge two observations together into a new WorldModel.
Besides being responsible for keeping the latest WorldModel it also contains functions to retrieve the state of an entity before the last observation of the entity. This can be needed for example when the door has been observed before, but is not present in the current WorldModel. Then the location of the door can be found using the last observed state of the entity.

4.5.2 iv4XR environment

The first component is the agent environment, it only contains the instance of NetHack and has an observe function to get its WorldModel. Since the NetHack object contained in the agent environment is only a representation of the internal state of the GUT. Each time the Java socket client receives a new game state from the Python socket server, the game state is updated. These updates only give updates on a single level, however the NetHack object also contains the latest state of the other levels and contains the previous game state as well.

4.6 Agent functionality

Each part of the program architecture is briefly discussed, the aim of this section is to provide with some information about the agent implementation in iv4XR. The agent implementation allows for navigation, interaction, and actions within NetHack. Several agent functionalities will be explained.

4.6.1 Navigation

The iv4XR framework contains different implementations of interfaces for navigation. Dijkstra's and A* algorithm have portals between the layers of the map. Multiple layers, each layer being a single 2D level, represent the 3D map of the complete game. Portals allow the player to an adjacent layer. The interface works for worlds with some conditions about the shape of the world. Each layer has one level, and the portals only move the player to an adjacent layer.
In NetHack the shape of the dungeon does not satisfy these conditions. Each layer in the dungeon can have multiple levels. And a player can take longer stairs, moving multiple layers. This makes navigation through the game more complex. A version of navigation which can handle the dungeon shape is Hierarchical Pathfinding A* (HPA*)[8]. This algorithm divides the level into chunks of a predetermined size and caches the shortest paths within the chunks to speed up future path queries. Since the dungeon can contain up to 81 levels, caching can make a performance impact.
An existing implementation of HPA*[22] written in C# was converted to Java using a combination of automated conversion and manual rewriting of the source code. It was only later discovered this implementation of HPA* did not allow dynamic changes to the map. An attempt was done at adapting the navigation graph during exploration, however this proved to be difficult because of caches.
In NetHack movement of the agent is restricted by static obstacles like doors and hidden tiles. There are also dynamic obstacles like enemies and boulders. Boulders can be moved around by pushing from the opposite side, unless an obstacle is on the tile the boulder would be pushed to. Moving a boulder can create a path for the player to explore more of the level. However, this was not implemented because of the difficult and in many cases there is an alternative path to explore past the boulder. The agent treats boulders as static obstacles to simplify pathfinding. Also, if the agent comes across an enemy, it will defeat the enemy before continuing its original navigation. Doors and hidden tiles are opened and discovered respectively before the agent will continue navigation. How navigation handles diagonal movement restrictions is discussed in the implementation chapter.

4.6.2 Item interaction

The agent is able to navigate to coordinates within the dungeon. However, when the agent needs to interact with items. Interactions with items can require several consecutive commands, selecting the item and which direction the item is used in. Because it does not make sense to update the WorldModel whilst specifying the action, the agent is able to send a list of commands to NLE. After these commands are all performed, the updated game state will be sent.
For an interaction, NLE will make sure the command is fully completed before sending the updated game state. This is done by continuously reading the next message until it indicates there is no next message, or when NetHack is waiting for new user input. This simplifies logic because the agent does not need to contain logic to deal with unfinished actions.

4.6.3 Object selectors

For interactions, the agent could need to navigate to a tile, entity, or monster. Each contains their respective coordinates, which are at random locations within a level. To simplify defining actions interacting with these different types of objects, selector classes were implemented using an interface. A navigation or interaction tactic can simply receive a selector as argument and given the current game state and WorldModel the tactic will lead the player to the given tile, entity, or monster.
Items do not have coordinates, however the same interface has been used to create a selector implementation for the player inventory.

4.7 Agent strategy

The agent strategy has several categories ordered based on priority. These categories are survival, interaction, and exploration. Survival is the most important because in play testing it is necessary for the agent to live. If the player is not alive, the game is stopped immediately and testing stops. The interactions are to pick up money, ammunition, or other items to help the player. The exploration section is last, so the agent will first interact with all necessary

items before exploring and going to the next level. This part is however essential to progresses the player towards getting to the final level.
In the following subsections, each of the categories will be discussed in more detail, with some examples to showcase how these are defined in iv4XR.

4.7.1 Survival

Survival has two different aspects to it. The agent needs to survive any enemies that try to attack it, but also there is a hunger state which needs to be managed to prevent the agent from dying of hunger. First, managing the nutrition levels of the player will be covered.

Nutrition

The player needs to manage nutrition, nutrition can be gained through a variety of methods. The least interactive method is by using a pray action, this action has a chance of giving nutrition to the player. The player may not perform pray actions very frequently since the player will then get punished. Another way to gain nutrition is by eating food, there are various types of food available in the game. Because the player can consume corpses of dead monsters. Older monster corpses have a larger chance to make the player sick and give other negative effects, so the agent will avoid eating old corpses. Food in the inventory of the player takes longer before it spoils, so it is the action with the least priority. The player can carry limited weight, so the best strategy is to first eat the nutrition that has the lowest nutrition per weight value. An example of how this can be programmed is included in Listing B.1.

Handling monsters

There are several aspects to take into account when dealing with monsters. Namely, whether the player has a melee or a ranged weapon, how close the player is to the enemy, and whether it should attack the enemy given the equipped weapon. Because additional information about the monsters is shared, the agent knows about each enemy. The first priority in the tactic is to equip a melee weapon and attack when the enemy is adjacent to the player. Switching a weapon does not cost a turn, so it is not wielded earlier. Otherwise, if the ranged weapon has all ammunition, the agent equips a ranged weapon. When an enemy is directly in line of sight across or diagonally, the agent will fire at the enemy. Attacking non-hostile enemies is avoided at all times to avoid unnecessary deaths.

4.7.2 Interaction

Interactions is the category used for gathering items potentially needed for later. Because the interaction has a higher priority, exploration will be paused until the agent has completed the interaction. A custom method demonstrated in Listing B.1, called interactWorldEntity, is used to reduce the work of writing new interactions. Only a selector and the commands to perform when the item is reached are required.
Examples of interactions are picking up potions, money, and ammunition for ranged weapons. Picking up ammunition for ranged items will help the survival of the agent when it gets into combat, whilst picking up health potions can make the player survive an attack by using it when the health is getting too low. So these interactions are useful to increase the chance of the agent surviving for longer. Since the agent has all the entities in the WorldModel when it enters a level, the agent will first pick up all relevant items before performing exploration. Whilst performing survival actions if needed.

4.7.3 Exploration

The agent explores by navigating to unexplored tiles. An unexplored tile is a tile with at least one adjacent unexplored tile. When the player reaches this unexplored tile, it potentially finds new unexplored tiles. Using this strategy definition until there are no unexplored tiles left, it will explore the entire level. To make the exploration efficient, the agent selects the unexplored tile with the shortest path.
Originally, the Manhattan or Euclidean distance were used when selecting the closest unexplored tile. However, this would get the agent stuck at times since it could get in a situation where moving to the closest tile would make another closest. So the player would constantly switch between multiple unexplored tiles, never reaching any of

them. Also, computing the closest tile through Euclidean distance does not guarantee it is the tile with the shortest path. This is because of obstacles in the level, because a level consists out of rooms and single-width corridors, the majority of the level are obstacles and so this can happen often.
Using the shortest path increases the complexity of the navigation calculations in each step. If exploring, it has to recompute the path lengths to each of the candidate locations and determine which one has the shortest path. Caching improves the speed of the calculations, but in addition some optimizations are made. If the Manhattan distance of the candidate is higher than the shortest path found so far, it will not compute the distance. Also, if the player and candidate are on two different levels. If the path distance between two levels is greater than the shortest path, the path length is not computed.

4.8 Verification using LTL formulas

During manual testing, the alpha testers use the interface of the game to perform their actions and verify their effects. The level they test on is different from unit testing, rather they test properties. LTL formulas can define properties the game not violate, and can automatically verify whether they hold.
The statistics at the bottom of the NetHack GUI have certain properties that should hold during the entire game execution. For example, the health should not turn negative or above the maximum health value. Since this does not depend on state changes, this can be enforced using assert statements. However, for some more complex properties, an assertion over the current game state is not enough. If we want to test the score is always increasing, the test requires the previous and current state and an assertion inside the NetHack code is required if this has to get checked at every step. The LTL formulas with the adaptation mentioned in section 2.3.2 are capable of performing these types of validations over each state without writing the check inside the object.
Even though, the example property is not complicated, an alpha tester can easily miss an inconsistency in this property. Since the numbers at the bottom of the screen change every turn, it is difficult to spot issues with these statistics. Also, after a lot of testing sessions, a tester will start paying less attention to the numbers since the tester assumes these remained correct for a long time. Defining this LTL property would remove some of the higher level properties of the game the alpha tester needs to verify, and can even detect violations of properties invisible to the tester. For example, when the LTL properties check each monster stays within the valid map coordinates and the monster suddenly glitches outside the map, this could go unspotted otherwise. And since it would test these numbers automatically in the background, it does not interfere with the game. The properties the LTL formulas will test are the following:

1. Score non-negative and does not decrease
2. Hp non-negative and not above maximum hp
3. Energy non-negative and not above maximum energy
4. The player only enters levels deeper into the dungeon
5. Hunger state can increase per step
6. Experience level can only increase, experience points only decrease when the level increases
7. Player turn cannot decrease
8. Player only traverses on walkable tiles
9. The player can only move to adjacent tiles

These LTL formulas are global, so it is surrounded by the G operator in the language. A formula definition of the first property is shown in Formula 2.2. These properties do not use complicated operators and instead aim at removing the need to visually check the statistics on the screen for bugs. So an alpha tester can focus on other aspects like gameplay. Interaction properties can also be checked, for example, items with charges like a camera with a camera roll can only go down one charge per use. These are small details, but this property is then verified in any gameplay and always reported when violated.

4.9 Coverage

The goal is to analyse coverage of the game logic, since NetHack is written in C++ and compiled using GCC, gcovr[5] is used to analyse the coverage. It is a popular coverage tool, allowing for collecting coverage over multiple runs. The results can be visualized in HTML and queried in XML or JSON format. The average will be taken over several play-through runs to get an indication of the variation as well.

The coverage is not collected over all the source code because there are utility files that are platform specific. These contain logic about drawing the map in the terminal. Since the aim is to focus on play-testing, only files containing logic for the play-through are included. This includes logic for level generation, monster behaviour, and shopkeeper interactions.

4.10 Mutation testing

For mutation testing, a source code mutation tool was selected. During testing, we need to easily select a mutation and know which mutation is being tested using scripts. There are methods that optimize the compilation time and integrate with the test library in C++. However, there would give less control over our testing since the test suite would also need to be included in the compiled code. The selected tool is SRCIROR[19] because it generates new source files with the mutations and can perform different type of mutations, listed below.

- Constants mutation
- Numeric operators
- Boolean operators
- Binary operators
- Conditions in 'if' and 'while'

The configurability of the tool, containing setting which lines are mutated, is another reason for choosing SRCIROR. This is useful for NetHack since some files contain thousands of lines, if only one method needs to get mutation tested it does not generate mutations in unrelated functions.

4.10.1 Repeatability

Maps, monsters, and entities are generated randomly, and a level is generated when the player enters it. So if the agent strategy would enter the next level at a different step, the rest of the game would be completely different. Because mutation testing performs the test suite multiple times, each time using the source code for a different mutation, the agent needs to be able to repeatedly reach the same state within the game. Otherwise, if the completion percentage for a test is below 5%, the expected number of test runs to test 30 mutations is over 600.
To avoid this issue, it was attempted to separate the random number generator from level generation, however it was not successful. So instead a replay file method was implemented to repeat a run exactly. This ensures that even when further development changes the tactics of the agent, a run for mutation testing would still work. The starting seed, information about the character, and actions are saved to the replay. This was created using the logger class to write the actions to file.

Chapter 5

Implementation

In this chapter, concrete implementation information will be given. By explaining how the game state is captured in Java and what improvements to the iv4XR framework have been performed. In total, the implementation of the socket server is over 1300 lines to communicate the game state. Interpreting this information in Java and attaching an agent to it is over 14000 lines of code.

5.1 StepState and GameState

Messages sent through the socket are in the structure of the Gym environment, a collection of arrays. This collection of arrays are put in a more logical structure called step state. An example is the player inventory, information about each property of the item are received in separate arrays. The logical structure is shown in Listing 5.1.

```
public class StepState {
  public Stats stats;
  public Player player;
  public Symbol[][] symbols;
  public Tile[][] tiles;
  public List<Monster> monsters;
  public List<Entity> entities;
  public String message;
  public boolean done;
  public Object info;
}
```

Listing 5.1: *The definition of the StepState class, the stats and player properties contain the statistics about the game shown at the bottom of the screen. Both the array of symbols are for drawing the level and the array of tiles is about the surface of the level. The monsters and entities are lists because the number of them is variable. Finally, there is some extra information at the bottom. A step state is created from each NetHack update.*

Because the dungeon can contain 81 levels, another class is defined to contain the game state. An observation only contains information of a single level. In the GameState class, multiple levels are combined into one object using some logic. This object contains navigation as well, which is expanded on next.

5.2 Navigation improvements

In the original iv4XR implementation, the tiles on which players could move in the Java game representation had to be copied over. Since iv4XR only copied information, it should be possible to use pathfinding on the Java representation itself. The author believed there was an approach that allowed for faster development and a reduce duplication. So the pathfinding was adapted to avoid creating a mirror map representation in the iv4XR agent state. The existing and improved implementation will be explained.

5.2.1 Existing implementation example

In this example implementation how navigation is performed in iv4XR, it makes use of subclassing. The main Tile class is not an obstacle. A subclass of Tile, called Obstacle, is the class under which all other Obstacle classes are created.
The duplication for the navigation in the iv4XR agent state is shown in Listing 5.2. There is one type of obstacle ("Wall") and tiles (""). Other types of objects are handled by using the Door class. Which can dynamically block the agent from moving over it.

```
switch (type) {
  case "WALL":
    multiLayerNav.addObstacle(new Pair<>(mazeId, new Wall(tile.x, tile.y)));
    break;
  case "":
    multiLayerNav.removeObstacle(new Pair<>(mazeId, new Tile(tile.x, tile.y)));
    break;
  case "MONSTER":
    // Monsters not represented in the navigation
    break;
  default:
    // Representing potions, scrolls and shrines as doors that we can
    // open or close to enable navigation onto them or not:
    multiLayerNav.addObstacle(new Pair<>(mazeId, new Door(tile.x, tile.y)));
    break;
}
```

Listing 5.2: *Depending on the type of the tile in the example game, there is an obstacle added or removed from the navigation object. The door is an obstacle, but it can be in an open state where the agent can move through. For example, a shrine blocks the player from opening until the correct scroll is used on it.*

Because there are only three types of tiles in the navigation graph, the implementation is concise. If a game has more types of tiles, additional information would have to get copied. In NetHack there are 20 types of tiles, some with special interactions depending on the state. For example, doors can sometimes only be moved to from one of the four directly adjacent tiles. But if the door is destroyed, the player can move to it from a diagonally adjacent tile. To allow for these differences and avoid a complex structure with subclassing, an implementation using interfaces has been proposed.

5.2.2 Improved implementation

Similar to information the previous implementation required, the proposed navigation implementation requires information about which parts of the map are obstacles and which parts the player can use for navigation. The new Tile class only contains the essential information needed in any implementation of the class. The definition is shown in Listing 5.3.

```
public abstract class Tile implements Serializable {
  public CustomVec3D loc;
  public final CustomVec2D pos;
  public boolean seen = false;

  public Tile(CustomVec3D loc) {
    this.loc = loc;
    this.pos = loc.pos;
  }
}
```

Listing 5.3: *The abstract Tile class definition is kept minimal so if another game needs to get tested there are no unnecessary properties. The CustomVec2D type is the two-dimensional version of CustomVec3D. These vectors can only contain integer coordinates.*

A concrete implementation of this abstract Tile class only needs to implement an interface in order to adopt its functionalities and let it get handled by the navigation class. Several interfaces the NetHack implementation uses are listed here.

- **Walkable:** A tile that can be walked over by the agent, can be a dynamic obstacle that is only walkable at times, an example is a door
- **StraightWalkable:** A variation of Walkable which only allows for moves from and to directly adjacent tiles, instead of also diagonally adjacent tiles
- **Climbable:** This tile allows the agent to go to another level, stairs or portals
- **Secret:** In NetHack there are some hidden tiles which can be discovered through searching
- **Viewable:** This tile can be part of the view cone of the player
- **Shop:** A tile that can be part of a shop

An implementation of the navigable assumes the level to be a rectangle and contains a 2D array of Tile objects. Another GUT in Java would then first have to make the surface tiles extend the Tile class, which only requires an assigned coordinate in 3D space. Information on whether the tile has been seen by the agent is also given. Additionally, this surface tile would implement the Walkable interface. Afterwards, the new implementation will be able to automatically builds a navigation graph with a node for each walkable tile. Finally, edges between adjacent walkable tiles are created. Using this approach, a developer avoids having to create special classes and duplicating the tiles in iv4XR to handle navigation.

5.3 WorldModel

In section 4.5.1 the content of the WorldModel has been explained. In short, it contains a list of entity information and their previous state. An improvement to the WorldModel is being proposed because the entities in the game are duplicated for the WorldModel and accessing properties is without checks whether those properties are set before accessing. Also, players, items, and monsters are all converted to a WorldEntity so it each time a string match is performed to filter the list to only contain monsters. The existing and improved WorldModel is presented in the coming part.

5.3.1 Existing WorldModel

The list of entities is a list of WorldEntities. The WorldEntity implementation contains some properties not each game will make use of. For example, in NetHack there is no velocity a WorldEntity can have. A game could have several entity types with additional properties. In iv4XR, an arbitrary number of properties can be added using a hashmap. A string is given as key and the value can later be retrieved. In Listing 5.4 the process of creating a WorldEntity is demonstrated.

```
Monster m = (Monster) e;
WorldEntity we = new WorldEntity(e.id, "" + e.type, true);
we.position = new Vec3(e.x, 0, e.y);
we.properties.put("maze", e.mazeId);
we.properties.put("hp", m.hp);
we.properties.put("ar", m.attackRating);
we.properties.put("aggravated", m.aggravated);
```

Listing 5.4: *This is an example of converting a monster to a WorldEntity. After setting the position, other properties are set in the hashmap. For a player, another definition has to be written because that object has different properties than the monster.*

The WorldModel contains some functions to easily retrieve a property from a WorldEntity. However, using a hashmap has multiple disadvantages. It creates opportunities to have a mismatch between the properties set in the hashmap and the properties that retrieved for later use. When the entity representation in Java is adapted by adding or removing a property, logic needs to get changed for copying the player, monster, or other entity to the WorldModel. Additionally, it is not known whether all uses of that property are changed properly. It would be beneficial to make use of the compiler and interpreter to keep track of existing properties and avoid manually adapting the logic to convert an object to a WorldEntity.

5.3.2 Improved WorldModel

Similar to navigation, an interface is used rather than using subclasses. An entity in the game representation of Java might already be a subclass, so then it would not be possible to use it in the WorldModel. The IWorldEntity enforces each entity has a timestamp and an ID to uniquely identify the entity. The initial improvement would have a list of IWorldEntity in the WorldModel. This would not give all the desired results when developing the iv4XR agent state and strategies. Then the WorldModel has a list of IWorldEntities and each use a cast is required to use read the entities in the WorldModel.
The solution uses generic types instead, the WorldModel class is then defined without a specific type. The definition 'WorldModel<WE extends IWorldEntity>' where 'WE' only refers to the type instance. It is enforced that the type instance should have the IWorldEntity interface implemented so the WorldModel has all the information to merge with another WorldModel. Since the player is a special type of entity that is accessed, often it is implemented separately. So the final class definition is 'WorldModel<P extends IPlayer, WE extends IWorldEntity>'.
This new implementation has improved the checks performed by the compiler and the ease at which new properties can be added and removed from objects. Because the exact type of the entity object is defined before compilation the methods have the exact type, then when a property is removed from the entity object of the game it will show an error if the property is being used.

5.3.3 Existing WorldEntity history

Initially, the responsibility of retaining a previous instance of an object was in the WorldEntity object. It retained a reference to the previous object state. This is so a strategy can use the last known state of the entity to find it in the game. If currently the agent does not observe the entity, it could navigate instead to the last known location. The timestamp of the observation is kept and also an ID to uniquely identify the items.
Iv4XR should have the responsibility to handle the history of the entities. An entity is not supposed to have knowledge about its previous state and keep track of this previous state. So this has been changed as well.

5.3.4 Improved WorldEntity history

In the newly proposed implementation, instead of the WorldModel containing a list of the 'WE' type, a list of the class called WorldEntityRecord is created. This class holds a reference to both the current and previous state of the 'WE' type. The definition for the class is shown in Listing 5.5.

```
1  public class WorldEntityRecord {
2    public WE current = null;
3    public WE previous = null;
4
5    public WorldEntityRecord(WE current, WE previous) {
6      this.current = current;
7      this.previous = previous;
8    }
9
10   public WorldEntityRecord(WE current) {
11     this.current = current;
12   }
13 }
```

Listing 5.5: *The definition of the WorldEntityRecord class. WE is a generic type which has to be an instance of IWorldEntity. If only a current WE is provided it the first record of the entity.*

Using this approach, it is much clearer the WorldModel keeps two states of each entity and the entity class itself has lost responsibility over keeping track of the previous state. A use of these two states is checking whether an item durability has gone down. Using the 'equals' method, iv4XR can determine whether a new instance of the object has been created and will update the record if a change is detected. The exact same approach has been done for the player type in the WorldModel.

Chapter 6

Results

In this chapter, we aim to answer the research question whether iv4XR could be used to effectively test the complex game NetHack. Several smaller research questions are formulated to help answer the main research question. To test the effectiveness iv4XR can reach, an implementation was developed to automatically play NetHack. Using coverage results and mutations score, it is assessed how effective the iv4XR agent is at testing the GUT. Additionally, LTL properties were created to perform testing more in line with the level at which alpha testers verify a game. Lastly, some improvements are listed for the iv4XR framework which can make future implementations of an iv4XR agent easier.
The results and answers to the research question will be provided in this chapter. All the results have been gathered on a Linux machine, Intel(R) Xeon(R) X5647, 12 GB RAM.

6.1 RQ 1: testing effectiveness results

To analyse the testing effectiveness, the code coverage of several runs is collected. Code coverage will be compared to a very sophisticated bot implementation for NetHack. Also, mutation testing results will be presented and analysed.

6.1.1 RQ 1a: code coverage results

The selected coverage tool is able to collect line coverage and branch coverage results. We will only look at line coverage since both the coverage types show similar trends. Before looking at the coverage results, the baseline against which the agent implementation is being compared to and the comparison method will be discussed. Afterwards, the line coverage will be shown.

Coverage baseline

The aim is to compare the code coverage results with another implementation that can perform a play through in NetHack. NetHack Challenge bot implementations did not aim at completing the game but tried to maximize the score. This resulted in some implementations that aimed at surviving as long as possible at a dungeon level. The agent would then continue fighting the monsters instead of performing a proper play through.
The coverage results of iv4XR will be compared with the coverage of BotHack[24]. BotHack is a good comparison since it is a bot implementation developed by a person as a side project over years. Using inspiration from already existing bot implementations. It is a very strong competing automated bot implementation and has been able to complete the dungeon at least once, creating an indication for what a play-through can achieve.

Coverage comparison

BotHack and NLE use different versions of NetHack. BotHack was developed on a much older version, namely 3.4.3, so the number of lines per file are different from the 3.6.6 version NLE uses. To be able to do a side-by-side comparison, the files of both versions are ordered on number of source lines per file. Even though the file

sizes have increased, it has increased evenly. So, trends found when analysing the coverage of BotHack can be compared to the iv4XR agent implementation.
The percentage coverage will also be shown to give a more comparable graph and have more insight in the coverage on a file by file basis. The standard deviation is also included to show how much variance the play through creates.

Line coverage iv4XR

The line coverage results for the iv4XR agent is shown in Figure 6.1. The results are averaged over 10 runs.

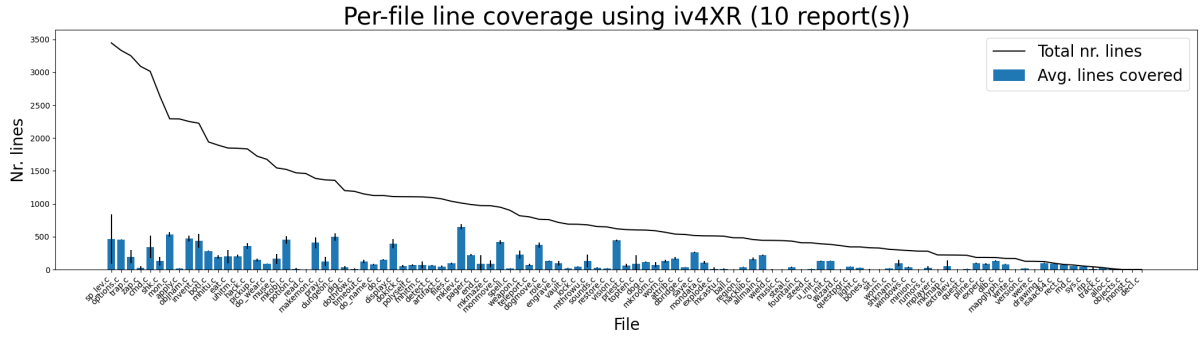


Figure 6.1: *Shows the line coverage of the iv4XR agent based implementation. The file names are sorted on size, the total size is indicated by the line. The total number of lines in NetHack is 94348 with 13398 lines covered. This equals 14.2% of the source code. This is a low coverage percentage but expected since NetHack has very many interactions and character specific logic.*

To give more insight in the variance Figure 6.2 is shown, it displays the coverage in percentages for each file. If a small file has a high variance, this will be more visible.

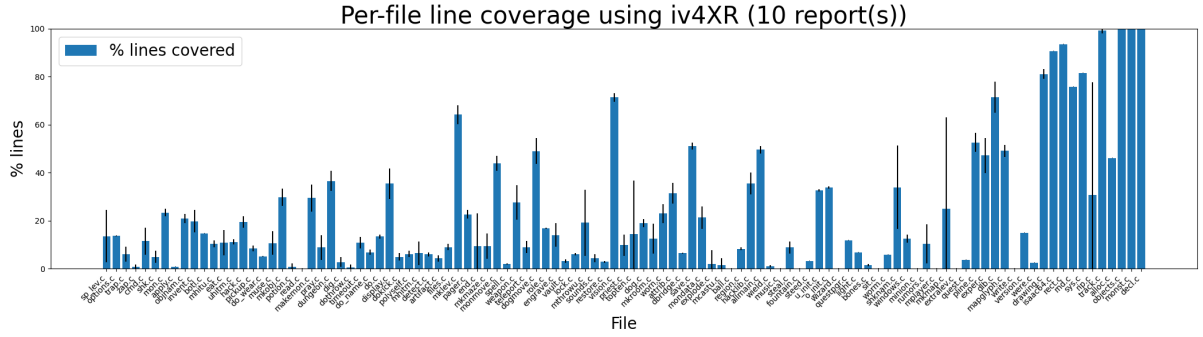


Figure 6.2: *Shows the line coverage of the iv4XR agent based implementation in percentages. The variance between runs is often small. There are some files with higher variance, example of which are: sp_lev, topten, shknam, mthtrowu, mkmap, and rip. Some files with low variance are botl, vision, save, o_init, u_init.*

Looking at the coverage results, some files do not have any coverage. The majority of the files have low variance, however there are some with high variance. File sp_lev is responsible for creating special levels, and mkmap is used as the initial map to create this special level on. These special levels are often after a couple layers, so when the agent does not survive for long, this logic is not being triggered. This also holds for the shknam which contains logic for handling the shopkeeper creation, if there are no levels explored with a shop then this figure will not be created, so the logic will not be used. The files: topten, end, and rip have a high variance since when the agent gets stuck in the level the final screen with the cause of death will not be displayed. Lastly, the mthtrowu file handles the scenario where the player is being attacked by monster with ranged weapons, this can sometimes not be executed.

The files with low variance are the init files because this only runs once at the beginning of the play-through and so does not depend on how the player gets in the game. The logic to compute the vision is performed at each step for the player, because a play through does average a few hundred steps the vision logic is executed many times so most is executed at least once. Even though there is no explicit saving of the game the logic in the save file is partially covered and has little deviation because the intermediate saves the game performs during a gameplay are frequent, it saves monsters, entities, and levels, in general, so it is not impacted by special levels or a specific type of monster or weapon.

Line coverage BotHack

The coverage of the BotHack is shown in Figure 6.3. Also for BotHack the results are averaged over 10 runs.

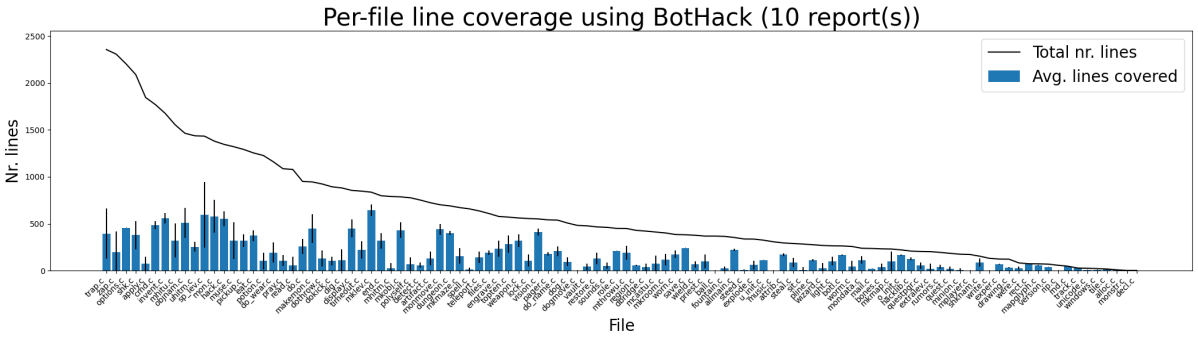


Figure 6.3: *Shows the line coverage of BotHack. The file names are sorted on size, the total size is indicated by the line in the plot. The total number of lines of NetHack is 65469 the average number of lines covered is 17189 which is equal to 26.3% of the total amount of code.*

The average coverage of NetHack using BotHack is nearly twice as high, this is mainly because NetHack logic being covered is very dependent on how much far the agent gets in the play through. There are many interactions and unique monsters and items which have their own logic. And special levels are only created when the agent comes across them. If the agent gets further into the game, it is more likely to come across these unique enemies, items, and levels and trigger the logic.

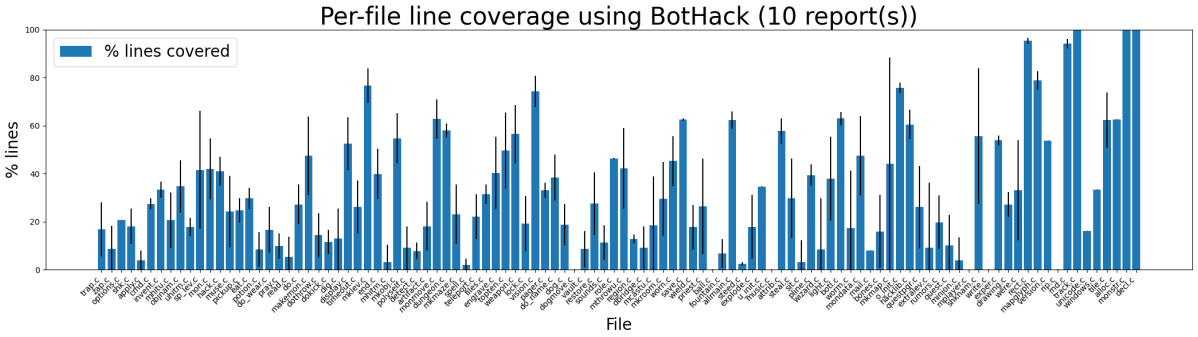


Figure 6.4: *Shows the line coverage of BotHack in percentages, note the variance is much higher than the iv4XR agent.*

The variance of BotHack is higher, meaning there is a greater difference in coverage between runs. BotHack is able to get further in the game because the overall tactic is much more sophisticated and makes use of many items it finds. This gives the bot a higher chance to get further in the game. Whilst collecting the results for BotHack there was a run of around 140k steps, unfortunately it was not possible to collect these coverage results since there was no indication it would finish within hours. The longest run included in the results took 1 hour and 55 minutes. This run had 43.6% coverage, the highest of the results. The shortest run, taking just under 1 minute and 30 seconds, only resulted in a total coverage of 18.6%. This makes sense because the more a program is run, the more code gets executed.

Reflection on RQ 1a

The research question to answer is what code coverage can be achieved. The code coverage achieved by iv4XR agent-based testing is 14.2% compared to 26.3% BotHack is able to achieve. Both these code coverage percentages are low compared to the standard of 80% code coverage in conventional software. However, collecting a high coverage of NetHack is very difficult. And so it expected to be much below the standard code coverage percentage for software testing.
NetHack is a complex game with many steps to complete, many items, interactions, and special levels exist and the automated play method needs to perform a long play through to have the possibility to perform actions with the items. Also, there are several different classes with different interactions in specific scenarios. So it is not possible to reach all code logic in a single play through of the game. More sophisticated agent behaviour will improve the code coverage, but to significantly increase the coverage percentage, a dedicated effort is required by test specific parts of the game. Many of the possible interactions are not required to finish the game, so the tactics have to include many interactions with otherwise unused items.

6.1.2 RQ 1b: mutation results

The results of mutation testing is the mutation score. The original code will be shown in this chapter, however the table with all the mutation results will be included in the appendix. Patterns of mutations that were not detected are discussed. The example is logic for a camera entity. The camera is a standard item for the tourist character, and it can also be spawned randomly in a level.

Generated mutations

To show what mutations the tool generates a short version of the logic will be given and mutations to this code are shown. A shortened version of the code is in Listing 6.1. This part is to give an idea of the mutations of a small part of source code.

```
STATIC_OVL int
use_camera(obj)
struct obj *obj;
{
    if (...) {
       ...
    } else if (u.dz) {
        You("take a picture of the %s.",
            (u.dz > 0) ? surface(u.ux, u.uy) : ceiling(u.ux, u.uy));
    } else if (...) {
        ...
    }
}
```

Listing 6.1: *A snippet of the logic to use a camera item. This part of the logic handles making pictures above or below the player. When the player aims up or down $u.dz$ will be non-zero so the statement is entered. Depending on the direction, it will show a message what is visible on the ceiling or on the floor.*

Mutations of this small section of code are for example changing the guard of the if condition by inverting it with $!u.dz$. Two other mutations it performs is changing the constant 0 to 1 or -1. The last type of mutations is replacing the '>' with other numeric operators. It replaces it by '!=', '<', '<=', '==', and '>='. Almost all of these mutations will change the resulting behaviour of the camera. From the message after using the camera, the test will be able to detect the mutations.

Camera entity

Because the tourist starts with this item, there is no tactic needed for this test. So this mutation test does not have to complete a number of steps before using this method. The logic of the camera in NetHack is shown in Listing 6.2. The code contains a lot of conditionals for different uses of the camera. When testing the camera, the aim is to test each interaction. Using a general play through would be unlikely to test this code fully. It might not even use the camera when it finds because it does not need to use it.

A reason why this entity functionality was selected to be tested is because it has a lot of different behaviours, but it is easy to verify whether the interaction had the expected outcome. By parsing the message after each action, it is clear which part of the code was triggered. Any mismatches between the expected and resulting message will fail the test and kill the mutation. Additionally, the behaviour of this method is largely deterministic, there is only one interaction when it is a cursed camera where it can have non-deterministic behaviour. Other parts of NetHack contain more random behaviour.

```
STATIC_OVL int
use_camera(obj)
struct obj *obj;
{
    struct monst *mtmp;

    if (Underwater) {
        pline("Using your camera underwater would void the warranty.");
        return 0;
    }
    if (!getdir((char *) 0))
        return 0;

    if (obj->spe <= 0) {
        pline1(nothing_happens);
        return 1;
    }
    consume_obj_charge(obj, TRUE);

    if (obj->cursed && !rn2(2)) {
        (void) zapyourself(obj, TRUE);
    } else if (u.uswallow) {
        You("take a picture of %s %s.", s_suffix(mon_nam(u.ustuck)),
            mbodypart(u.ustuck, STOMACH));
    } else if (u.dz) {
        You("take a picture of the %s.",
            (u.dz > 0) ? surface(u.ux, u.uy) : ceiling(u.ux, u.uy));
    } else if (!u.dx && !u.dy) {
        (void) zapyourself(obj, TRUE);
    } else if ((mtmp = bhit(u.dx, u.dy, COLNO, FLASHED_LIGHT,
                            (int FDECL((*), (MONST_P, OBJ_P))) 0,
                            (int FDECL((*), (OBJ_P, OBJ_P))) 0, &obj)) != 0) {
        obj->ox = u.ux, obj->oy = u.uy;
        (void) flash_hits_mon(mtmp, obj);
    }
    return 1;
}
```

Listing 6.2: *Logic used for the camera entity in NetHack from the apply.c file. The conditionals from top to bottom are for the following scenarios. 1. Player is underwater 2. The direction the camera is being used is invalid 3. The camera has no charges left 4. Camera is cursed 5. The player is swallowed by a monster 6. Picture being taken of floor or ceiling, 7. Picture being taken of player 8. Picture being taken of a monster. The method rn2 is to randomly zap the player half the time when it is trying to take a picture with the camera.*

The mutations and results can be found at Table A.1. The mutation score, percentage of mutations killed, is 56.9% overall. A part of the logic of this method was not possible to test using the agent. The agent was not able to reach a level with water after giving it many tries. So, the logic for using the camera whilst underwater is not covered. Also, it was not possible to reach a scenario where the player was swallowed by a monster or the non-deterministic behaviour of a cursed camera.
The return values of numeric mutations were not detected. These do not have an effect on the game, so it is not possible to test these mutations. And being able to test the cursed camera item would improve the mutation score.

Reflection on RQ 1b

The research question we want to answer is what mutation score can be achieved. The mutation score that is achieved for this camera example is 56.9%. It is not possible to detect all the mutations even with perfect strategy created. The majority of mutations are detected. To increase the mutation score, further control over the random number generator and getting further into the game, so the agent can perform the interaction.

6.1.3 Reflection on RQ 1

When answering what effectiveness testing can be achieved using iv4XR, the code coverage percentages indicate low effectiveness. A percentage of 14.2% is far lower than the standard for code coverage when testing software. Improving agent tactics will increase the coverage percentages because the agent will get further. However, having a high coverage percentage does not mean bugs will be found. The bugs it is able to find are unexpectedly terminations.
Mutation testing is a method to quantify how effectively the testing software is able to find bugs in the game. When wanting to test the mutation score, it was fastest to perform a sequence of actions which will target testing of a method. Making it similar to unit testing. Using this approach, a mutation score of 56.9% was reached. There is the possibility to increase this mutation score when the agent strategy is improved. The use of iv4XR to test a GUT is by having the agent reach a desired state in which the developer wants to test. Then perform interactions with the item to robustly test all item interactions in the game.

6.2 RQ 2: facilitating testing using LTL

As stated in section 2.3.2, the previous game state is also saved in the agent state to define LTL properties which check state transitions. An example of using the previous state is ensuring the score never decreases during the run. Properties which are checked are listed in section 4.8.
These 9 invariants have been formulated to check aspects of NetHack. Invariants should hold during the entire run. The last property, to check the player only moves to adjacent tiles, is shown in Listing 6.3.

```
public static Predicate<SimpleState> adjacent() {
  return ((SimpleState S) -> {
    NetHack netHack = ((AgentState)S).app();

    // No previous state
    if (netHack.previousGameState == null) {
      return true;
    }

    CustomVec3D prevLoc = netHack.previousGameState.player.location;
    CustomVec3D loc = netHack.gameState.player.location;

    // If the player has not moved or has changed the level
    if (prevLoc.equals(loc) || prevLoc.lvl != loc.lvl) {
      return true;
    }

    return CustomVec2D.adjacent(prevLoc.pos, loc.pos, true);
  });
}
```

Listing 6.3: *Checks whether the previous position of the player is adjacent to the current position. A position is adjacent if it is directly or diagonally adjacent to the previous position. If the player has moved between levels, it does not check for adjacency.*

The results, whether the LTL formulas are satisfied, are collected over runs. Iv4XR evaluates the formulas and monitors whether any of the formulas are unsatisfied. When any or with multiple unsatisfied LTL properties, a check is done to inspect the cause of the relevant properties. Since the implemented LTL properties are invariants, it can be checked automatically at which point in the play through the property was violated and give a report on when the property was violated.

6.2.1 LTL properties results

The agent reported 3 violations of LTL formulas during a longer play through. The three violated properties are moving only to adjacent tiles, as defined in Listing 6.3 and non-decreasing score and level number. The non-decreasing level number is a property which is not expected to hold over a longer play through because the agent can freely move in-between levels. It is expected the agent will sometimes go back to the previous level to continue exploration or to collect food or items.
The score property was expected to be valid during a play through. However, the score in NetHack does not correspond to the score that NLE shares. NLE shares the result of a function to support learning algorithms. So it contains a penalty when there are no changes to the state of the player. This penalty caused the score to drop, this was detected by LTL.
The third LTL property detects possible invalid moves of the player. This violation was caused by a teleporter trap NetHack contains, when the player steps on this trap it gets teleported to a random position within the level. It is clear this causes the property to be unsatisfied. Because it was not possible to find the trap information in the NetHack internal state, so the definition could not be extended to handle teleporter traps.

6.2.2 Reflection on RQ 2

The research question to answer is whether LTL can facilitate testing. The defined properties used to gather results did not have a high complexity. However, they managed to find flaws in some assumptions made about the game logic. This shows it is possible using LTL properties to find properties that are unsatisfied during a run. More long term planning with quests and puzzles could make use of more complicated LTL formulas and show more benefits of using LTL formulas for games. Since the properties that can be defined in LTL are similar to the level of testing of alpha testers. It potentially can assist them during their sessions.

6.3 RQ 3: what improvements can be made to iv4XR?

During the thesis, an entire iv4XR agent and the connection to an existing game was established. During this process, some observations of aspects that could be improved about iv4XR were made. These improvements could make it easier for future projects to connect the framework. Some of these improvements have already been made and are already discussed, including the implementation, in chapter 5. Other changes that were not done but could be improvements for the future will now be explained.

6.3.1 Conditionals

A proposed improvement has to do with predicates. Predicates are used to enable/disable actions. However, the current implementation is prone to duplication. If multiple actions have the same condition, for example if the agent is hungry, each of the actions that aim at resolving this hunger state will each check whether the player is hungry. The code is provided in Listing B.1 and shows how in each action this conditional is separately defined. A few solutions will now be presented, and the proposed solution will be explained.

Add goal structure

A solution for the problem can be to add a goal structure. When the goal structure is active, until the 'agent is hungry' goal is not reached, it will perform actions to resolve this. This approach however causes duplication in another way. Whilst the agent would perform the goal structure to resolve the hunger state until completed, there must be other survival tactics in the goal structure as well because enemies pose a greater immediate threat. Their actions could kill the player. So the survival tactics are duplicated and put inside the new hunger resolving goal structure. The survival tactics, and potentially other necessary tactics and actions, will also have to be included. Additionally, when changes are made to the main agent tactic, this also needs to be changed in the other goal structures. So this would increase the amount of work to maintain the goal structures in an agent implementation.

Add multiple 'on' predicates for actions

Currently, an action can have one predicate to determine if it is enabled or disabled. If multiple predicates can be added to a single action, a predicate definition can be added to multiple different actions as separate guards. Since this only reduces the length of the duplication, it does not solve the root problem. It can also reduce transparency because it is not clear which conditions are attached to the action when it is passed around as an argument.

Add a condition structure for a tactic

To avoid having to specify whether the player is hungry in each action. The condition can also be used at a tactic level. If the evaluated guard of a tactic is false, then the actions in the tactic are not evaluated. How this would look in practice can be seen in Listing 6.4.

```
goal("Main agent explore goal")
  .toSolve(
    (Pair<AgentState, WorldModel<Player, Entity>> proposal) ->
      proposal.fst.app().gameState.done
    )
  .withTactic(
    FIRSTof(
      // Survival
      TacticLib.attackMonsters(),
      TacticLib.resolveHungerState().on_((AgentState S) -> S.app.gamestate.player.hungerState.
          wantsFood()),

      // Item interactions
      ...

      // Exploration
      ...
    ));
```

Listing 6.4: *A shortened version of the agent goal structure is shown. When a condition to a tactic can be added, it could be implemented like this. The same structure as guards for actions can be added to a tactic using the on_() method. In this case, it would not enter the tactic when the agent is not in a hunger state.*

This solution is similar to how guards are implemented for actions, and it prevents duplication over multiple action guards. This is the proposed solution to improve the tactics in the goal structure of the iv4XR framework. The limitation of this implementation would be that it can only work without passing an object down like the $.on()$ function, but it is not clear where the object would be passed through within the tactic.

6.3.2 Logging actions

Iv4XR emits logging messages when the conditional of the goal has been reached and also when a goal has been aborted. These messages indicate when the desire, goal structure object in iv4XR, changes. During development, it is useful to have more transparency for actions of the agent. It is currently not always clear which action is selected in a step. During the development of this research, sound effects were used to quickly interpret what action was selected. For example, if the action had to do with exploring another, sound would play than when the agent navigated to an item or tile to interact.
To add more transparency to know which action is selected, it would be useful to have some way this information is logged and can be visualized. Since actions contain a name, no additional changes besides logging the selected action have to be done. Logging in iv4XR can be configured by the configuration file as well, so if the additional logging is not desired it can be turned off or disabled by changing the configurations.

6.3.3 LTL clarity

When violations of LTL properties are reported, it is currently unclear which LTL property was not satisfied. The index of the violated LTL formulas is reported, so it only depends on the particular order of the LTLs formulas when given to the test agent. It would be beneficial to attach names to LTL formulas, so it can be clear from the

violation report which LTL property has been violated during the run. Then the ordering of the LTL properties does not matter.
Additionally, an invariant which has to hold at each state can specifically report at which point in the play through it was violated. Then it is easier to debug an execution when a violation has been reported.

6.3.4 Reflection on RQ 3

To answer what improvements can be made to iv4XR, we have to look at the implemented improvements in chapter 5 and some proposed improvements presented in the results section. The implemented improvements are for navigation, WorldModel entities and keeping track of the previous entity state. In this section, several improvements have been proposed to reduce code duplication, clarity agent action selection, and make LTL properties more verbose.

6.4 Reflection on the main RQ

The main research question is whether agent based testing, using the iv4XR testing framework, allows for testing of complex games. The answers to the sub research questions show it is possible to create an agent able to play partially through a complex game. However, the coverage percentage reached is low for this complex game, and it requires a lot of additional effort to reach a reasonable coverage percentage of 43.6% like BotHack. The agent is able to play NetHack and may be used for reaching a state in which a unit test or other test can be performed to detect mutations or bugs in the source code. This could save time in creating a small test setup. LTL properties can test properties in the game to reduce how much an alpha tester needs to pay attention to. Improvements to iv4XR will make the process of writing an agent implementation faster and hopefully improve maintainability of an agent strategy.
Implementing an agent in iv4XR could not get high enough results or test enough parts of the code to test the complex game effectively. However, it can help developers and alpha testers by testing properties during a play through defined using LTL. And a simple definition of an agent can perform some automated testing. However, for meaningful automated testing, an extensive agent strategy is needed.

Chapter 7

Related Work

In this chapter, previous literature work relevant to the thesis will be presented and compared to how it relates to this work. The research will be grouped together into different parts. At first, automated game testing approaches will be listed. Then the verification methods used to verify the testing approaches. Finally, LTL formulas and their use in software testing will be discussed.

7.1 Automated game testing approaches

Many researches have been done which try to reduce the amount of manual testing required for the game industry. Because we are interested in automated game testing in scenarios where levels can change, we will not discuss testing through scripting. Other approaches which can handle changes to the level and perform automated game testing are listed below.

7.1.1 Random agent testing

This testing approach does not use the information provided by the environment in the decision process for their next action. These agents are used as a baseline in some research to compare their result against, like explorative ability[31, 50]. Such an agent is fast to develop, however their randomness limits explorative potential. When used comparing against other exploring methods, their performance is the worst. Human testers and other automated explorative testing perform better than the random agent. A random agent can only be used in small flat worlds, when there are 3 dimensions and a big world like in NetHack the random agent cannot explore the game enough for meaningful testing. Also, the random agent has limited ability to interact and test the surroundings because in many cases the player is required to be in an area to perform the action. The chance of getting to the location to perform a specific interaction is limited. It is unpredictable whether the action will be executed when the agent is in location because it is triggered randomly.

7.1.2 Model-based testing

Model-based testing is a more programmatic approach, in which the state of the game is modelled using a state machine. Test cases can be generated automatically, the goal of these test cases is to select a transition path which will reach a certain state in the state machine. A state does not have to be the final goal of the GUT, but could also be a state in which the player has died[20]. To perform this type of testing, additional work is required to create the model with states and state changes. This testing method has been used to test a Mario game.
Using this testing method, it is possible to automatically generate testcases to test all the states and state transitions. Without knowing the expected outputs of the automated test, a verifying the code can be done by generalizing the output results or checking the program does not terminate[47]. A complex game increases the size of the model and results in many transitions. Mapping out these transitions and different states was possible for a Mario game where the player can be alive or dead and there are only a few inputs in the game. However, in NetHack it is not clear what output an action has and how to come to a next state. This is also caused by the non-determinism which is a key part of the game, it is not possible to know beforehand whether going to the next level at the current turn

will send the player to a special level or give the player a level in which there is a particular item present. Because of this, NetHack is not a candidate game for a model-based testing approach.

7.1.3 Search-based testing

A different method for testing is search-based testing. It is used to automatically generate test data for black box systems. It uses optimization techniques like hill climbing, simulated annealing, genetic algorithms, and other methods to generate certain outputs that for example cover a certain execution path in the code[35]. Search-based testing has been applied in game testing during another iv4XR research[14]. A model is created of the game beforehand and then using search algorithms are used to generate a testing strategy to reach the goal within a maze world. This maze world consists out of many doors and buttons that can interact with multiple doors at once. Even though, the underlying model can be a black box, it is difficult for NetHack to make use of search-based testing. Non-deterministic behaviour within the game significantly complicates the search space.

7.1.4 Machine learning

Using Machine Learning (ML) approaches have popularized in games testing, and their use has increased over the years[23]. These ML implementations are able to learn strategies and, in some cases, out-perform human experts in performance. In addition, they have been deployed effectively with the focus on finding bugs by exploring many states besides finishing the game.
This testing approach has proven the ability to detect many types of bugs. Examples of which are: world navigation[18, 46, 49], game balance[46], visual bugs[39, 49], and stuck states[7, 18].
A large advantage of this testing technique is that the model only requires knowledge about the game state, combined with possible actions and reward function to guide the learning process, and is then able to learn a wide variety of games after spending time training. The effectiveness of ML approaches for NetHack has been tested during the NetHack Challenge. The implementations that used ML were outperformed by programmatic implementations using strategies and long term planning[2]. The created reward function made it a safe option for agents to stop exploring the dungeon further, and instead stay at a level where a lot of enemies spawn and continue killing enemies there. Because NetHack players can progress in different ways, for example by getting stronger items, getting XP from killing enemies, training statistics of the player, gathering additional food, it is very difficult to define a good reward function. Another difficulty is to define a function to evaluate how well the agent is doing in the current state. And it is difficult to understand whether it did well so far because each time the items the agent came across are very different.
When ML is trained to maximize the reward function. A reward can be given to try to perform an impossible move. The agent will do so in order to gain a higher score. But it can be difficult to then control how many times such an action is performed. And when a new bug has appeared and needs to get tested in the future the model needs to get trained again, or an additional model needs to be created. Making adaptations to the ML agent requires knowledge ML developers have. But for game developers who want to test a game, this can create a barrier to work with this technology.

7.1.5 Agent-based testing

The last approach that will be discussed is agent-based testing of games[43], this is a testing approach from the perspective of the player. The player, controlled by an agent, decides which action to perform. This method is more flexible than model-based testing and search-based testing because only a partial model of the level is required to perform testing. This partial model is extended during gameplay by new observations of the agent. Based on the belief state of the agent, it will decide what action to perform. This makes it possible for this testing method to deal with sudden changes within the game state during a play-through.
A larger time investment is required to perform this method of testing because there is no automatic method for the agent to learn how navigation works within the game. When this is done, there is very fine control over how the agent behaves. It can be quick to add or remove specific behaviours. With a framework like iv4XR, some of this initial time investment can be reduced by implemented standard functions expected to be similar for games in general.
Also, agent-based testing is able to perform short and long term planning[15] whilst ML algorithms only has a single goal to maximize the reward function. This short and long term planning is essential in a game like NetHack

because there are some special levels the agent can find in the dungeon which have a very different strategy to play through them because mechanics of the game change. A different strategy to handle the specific level can be used.
This is a good testing approach for NetHack because the situation for the agent can change quickly. Ensuring the model stays correct can be challenging[44]. Because NetHack a game is one long gameplay, the short and long term planning that is possible because of the BDI cycle can help with creating sophisticated strategies that allow the agent to survive. Finally, this strategy has been shown to be most effective for NetHack during the NetHack Challenge, and there have been NetHack agent algorithms in other languages using a goal and strategy approach which are successful at playing and even completing NetHack.

7.2 Game testing verification

To verify the effectiveness of a game testing approach, several different methods are used in literature. Namely, number of bugs found and achieved code coverage. Even though many specializations have benchmarks to test their implementations on[25, 28, 36], more specifically bug benchmarks[11, 33, 34]. It has been lacking behind in the game testing industry, which makes it difficult to compare effectiveness of the performed studies[30].
Whilst testing approaches using machine learning are used numerous times to find bugs in existing games[31, 50], other testing approaches often verify their method on a simplified game, sometimes specifically made for their research[41]. There has been a recent effort to create a central benchmarks for gaming testing[30] which contains 5 already existing games and different types of bugs.

7.2.1 Coverage testing

There are different types of coverage, the type of coverage collected in this thesis is line coverage. A classical method in software is to look at line coverage, sometimes in combination with branching coverage of an application. This is the percentage of lines, and for branching coverage the number of different execution paths, being executed by the program. However, research has shown that both these types of code coverage are not strong indicators of sufficient game testing[30]. Coverage is relatively easy to achieve by random testing in a small game with few actions. Whilst for larger games with more complexity it is not a good indicator[50]. From the same research, it is shown that state coverage is a good indicator. This coverage indicates the number of different states covered by the system, but it does not take unreachable states into account.
Because the internal state of NetHack is very extensive and because it is difficult to know which states are similar when everything is random each play through, it was not possible to perform state coverage. So even though, line coverage is not a good indicator for how sufficiently a game is tested. It still is used to get insights in how much of the game code is being run with the agent. Also, it is a standard approach in other researches, so it is able to compare this research to other papers using this testing measure.

7.2.2 Number of bugs found

Many testing methods in mentioned researches verify their effectiveness by the ability to find bugs[4, 18, 50]. These bugs can intentionally be introduced when developing the game to test on. Another way to introduce bugs is achieved by manually adapting the original source code with known bugs in previous versions of an existing game, and then verifying the testing approach is able to find those bugs.
In other researches, bugs are found in already existing games that were unknown beforehand. This is a good indicator the method is capable of finding real-life bugs. However, it is difficult to quantify its capabilities since it is not known how many have been missed that were present as well.
For NetHack it was difficult to manually inject bugs because the author is familiar enough with C++ to adapt the source code. And it was difficult to find which parts of the code were adapted previously to resolve a bug. The known bugs present in NetHack are currently mainly game mechanic bugs or very specific interaction because the game has been tested by testers and players have reported the bugs they have found, so the developers have solved all big bugs. So, expecting to find new bugs in NetHack makes it difficult to quantify the results, so this method of assessing testing effectiveness has not been used in this thesis.

7.2.3 Mutation score

Mutation testing has been researched and applied for several decades for software testing and is now well established. There are many mutation approaches that can be performed[21]. Ranging from unit level, to integration testing level. It is also possible to create mutations on the log[13]. In game testing literature, as mentioned in section 7.2.2, bugs are manually introduced into the game by adapting the source code. It is sometimes performed automatically[14] in order to test how many faulty programs would be detected. It is however not standard to perform mutation testing in games. This prevents the possibility to perform targeted mutations in code that are expected to be caught by the testing method. The advantage of this testing method, compared to the number of bugs found, is the ability to quantify the results and the robust testing on tens of automatically generated mutations of a single method. The mutation score also shows a pattern of which type of bugs the tests are not yet able to detect.
Because mutation testing is well established, there exist many tools for different languages to choose from. And there is configurability, for example it is possible to configure which lines are allowed to get mutated, so it will then create targeted mutations. Mutation score is not used often in game testing, this thesis tries to draw some attention to the validity of this effectiveness measure for a testing approach.

7.3 Testing using LTL

Other researches have explored using LTL properties to perform testing. Frameworks like QuickLTL[38] have been created to perform LTL property verification and testing. Even with non-deterministic behaviour, there is the possibility to test using LTL[3]. A good aspect of LTL property testing is that the properties do not need to be defined in the original source code.
Another research was discovered where LTL properties were used to perform automated game testing[48]. That research shows the processing needed to monitor the properties is minimal and linear, resulting in an efficient approach. Also, it was shown that bugs in the game were being found by the LTL formulas. Bugs are quickly and automatically discovered using LTL properties, since they can validate every run. So it can reduce the time it takes to find a bug in the game.

Chapter 8

Conclusion & Discussion

The current research aimed at investigating how the testing process for complex games could be improved using agent-based testing in the iv4XR framework. Specifically complex games because existing agent-based research often was performed on games made specifically for the research or small existing games.
The main research question was whether agent based testing, using the iv4XR testing framework, allows for testing of complex games.
An agent implementation in iv4XR was developed and connected to NetHack using NLE. The agent had a single goal and used a general survival, interactions, and navigation tactic to reach as deep in the dungeon before the agent got stuck or died. During the development process, various aspects of the iv4XR framework were improved. The performance of the implemented iv4XR agent was compared to a strong existing implementation called BotHack. The results showed BotHack was able to achieve a higher coverage in the game, also the maximum coverage of BotHack was high for a complex game. The mutation score of a test of the camera item was assessed. This score showed the agent was able to test the majority of interactions of the item and indicated it was possible to robustly test code in NetHack. LTL formulas were defined to test whether some global properties in the game would be satisfied in all runs. The violations of the formulas were correct information, analysing at which turn the violation of the property happened showed what caused the violation. Lastly, improvements to iv4XR reduced duplication of information, made more use of compiler checks, and moved responsibilities of navigation and entity history. This made working with the framework easier and more intuitive. Proposed improvements which were not implemented yet can help future research using the framework.
This research has shown that agent-based testing is not able to effectively test the complex game NetHack. This is partially due to the large code base of NetHack and non-deterministic behaviour. These made it difficult to reach a high code coverage, define complete tactics, or have control about what happened later in the run. However, using LTL properties to test properties in a game provided an automated testing tool working at the same level as alpha testers. But future research is needed to explore more of the capabilities of LTL within game testing and measure how helpful this is for alpha testers.

8.1 NetHack as GUT

This research was aimed at testing a complex game. The selected complex game is NetHack, however this is a game developed several decades ago and only played by a select group of people.
Reasons why this game was selected because of the NetHack Challenge held a few years ago which provided an extensive API. So, information about the state was available without having to change the source code of NetHack. Additionally, the world had a grid based map of squares. An implementation of navigation was already created for this type of map. The last reason that will be mentioned is the speed the game can run at. NetHack is written in C++ and heavily optimized, so as long as the agent implementation would be quick, play testing would not take much time.
A limitation of selecting NetHack is that it is not clear how the complexity compares to modern game. NetHack has mainly been developed open-source by multiple different developers over time, so NetHack is many lines of code. However, most lines contain game logic because the visualization method is ASCII in a terminal. Modern games, even when also turn-based, could have a different type of complexity than NetHack. The complexity of

modern games could be rendering, special effects, multiplayer communication, or other aspects than logic. Would agent-based testing also be useful if the complexity is different from game play logic.

8.2 Assessment of the agent

Three different assessment criteria were using during the research. Namely, line code coverage, mutation testing, and unsatisfied LTL formulas. In the related work, it was discussed that line coverage is not a strong indicator of sufficient game testing, especially in games with a high complexity. It was also made clear why a state coverage criteria was not possible for NetHack, but it is clear that line coverage is not a good indicator of how sufficiently NetHack is has been tested by the agent and BotHack.
The aim of this research was how the testing process could be improved. As currently, alpha testers are often used for game testing. Mutation testing gives a good indicator of how sufficient code has been tested and is an automated testing method of introducing bugs to a game. However, the tool introduced bugs at a unit level. How much of the effort of alpha testers will be reduced by effective mutation testing is unknown. This would need further assessment to verify.
Finally, LTL testing was assessed by whether it was able to find violations of properties during a run of NetHack. Even though the results were promising, and could with real properties that have to hold within a game be reporting on real bugs. These results were not produced by an empirical test of LTL properties. So future work, if possible also with more complex formulas, is needed to support this observation.

8.3 Iv4XR framework reflection

The iv4XR framework does provide a method to intuitively program strategy. Because the actions and tactics can be given clear names, it is possible to quickly understand the goal structure without having written the agent strategy, as can be seen in Listing 4.1. And the interfaces of iv4XR framework makes developing an agent easier.
The program architecture contains three different programming languages, where NLE (Python) acts as communication line between the Java client and NetHack. However, it caused an extra step of communication, and thus created extra overhead. When information shared from NetHack has to be increased, the following steps are need. The internal state needs to expose the extra data, NLE has to convert it to a message and send it via the socket, and finally the receiving client must parse the information and add it to the game state. Only after all these steps are updated, the agent will be able to receive the information. Because the game changes frequently during development, sharing the entire internal state could cause a lot of extra work.
The framework is useful because it makes it easy to develop an initial version of an agent. After sharing the game state, only a few extra steps are needed in iv4XR to make an initial agent version work. Commands need to be able to be sent to the game to perform actions, the navigation interface needs to get implemented, and some general helper functions to query the game for an item. Also, after defining some strategies making more helper functions, it is very quick to add new interactions in a tactic. Taking only a couple of minutes to extend the tactic with picking up health potions and using them when the player has low health points, for example.

Chapter 9

Future work

The goal of this section is to propose future research directions and propose improvements that can be made to the current implementation in order to improve the research or different research directions for automated testing of complex games.

9.1 Improve agent strategy

A lot of work was required to get an initial agent strategy to work for NetHack. All the parts needed to get adapted and implemented, from sharing more information about the internal state to the new navigation implementation. By improving the agent strategy, the agent will be able to reach higher levels and explore a larger part of the game and logic. The agent will have a higher chance of coming across an item that needs to get tested.

9.1.1 Improve current agent strategy

The current agent strategy has the 3 aspects, survival, item interactions, and exploration. The agent currently dies frequently by fighting enemies that are too strong. If it is possible for the agent to perform a ranged attack it does so, this improved the survivability. A way to improve survivability more is by giving the agent tactics to move specifically to squares from which it can attack an enemy with ranged attacks. Another aspect that could be improved about the agent is using stronger weapons it finds during the play through to perform more powerful attacks to enemies. Currently, the player only uses the initial weapon it has and any ranged weapon ammunition it finds during the game. Using potions and implementing logic to buy items from the shops can also help the agent getting stronger.
Using spells in NetHack is also something that would be helpful for certain classes of players. When the player gathers experience, there are spells that unlock. It is a challenge to extract this information from the internal state of NetHack since NLE only shares information about the map, not the visual menu of spells. These spells could however speed up regeneration and perform special powerful attacks. Currently, these mechanics in NetHack are unused.

9.1.2 Using goal structures

Iv4XR has the possibility to perform short and long term planning using goal structures. Further into the play through of NetHack the player can reach a special level. A special level can have different logic. The default goal structure, aimed at surviving and exploring, does not give the agent a method to complete a special level. Some logic could be added, specifically aimed at finding gear before a special level. And special goal structures can also be used to fight bosses in NetHack for example when fighting Medusa a mirror is useful and the agent would have to move differently to avoid her attacks.
Currently, the agent does not get this far into the gameplay in order to have quests, but this could be another use for using multiple goal structures. Completing these quests will reward the player, so it is useful to make completing a quest the goal temporarily.

9.2 Explore the usability of testcases preparation

A possible useful application of iv4XR and other agent-based testing is to create testcases automatically. When creating tests, setting up the scenarios for testing is a time-consuming process. In many cases, a minimal scenario is created to perform a targeted test. When a test requires an existing level, changing that level could then break the test. Each set of tests needs a different scenario, using agent-based testing to find these scenarios is a robust method. For example, if a run needs to be created where the agent has an item in the inventory and has to use it whilst being in the water. The agent can perform runs until it reaches this state. When it finds the state, it saves the path to it and then a suite of tests can be performed to test interactions. The tests only need to get adapted when these interactions change. So it can be valuable to see if this can be a solution to aid in game testing.

9.3 Assist testing using LTL properties

A combination between automation and manual testing can be using LTL properties for testing. During development of the program, assertions in the code were used to automatically verify properties in NetHack. For example, all discovered paths were verified, checking the start and end coordinates, making sure all locations in the path were adjacent and unique to avoid cycles, and ensure the path should is empty if the start and finish are the same coordinates. In a play through of 500 steps, this would automatically verify over 5000 different generated paths.
An application of this for methodology to game testing can also prove to be effective. When a play through is performed by alpha testers, properties could be verified through the use of LTL and reported back or logged automatically if the property is violated. An example of what can be tested is an obstacle state does not change until the player interacts with it. Or a locked door can only be unlocked or destroyed, after the door is destroyed it cannot be closed or locked. These tests can reduce the amount of work of the alpha testers because a part of the properties in the game are being checked automatically. A violation of the property would be guaranteed to get reported, so a bug is spotted as soon as possible. With increased automated testing, the alpha tester can focus more on other aspects of the game.

9.4 Simulating personality to improve testing effectiveness

Testing games using human like behaviour has shown to be more effective in finding bugs or problems in games than synthetic bots[4]. BDI can be used to create human-like agents[40] because of the practical reasoning, which is intended to be similar to human reasoning[9]. What agent-based testing provides is a method to have fine control over the tester and actions. These different personalities could be utilized to increase the effectiveness of testing without needing several models like machine learning approaches[50].
PathOS is a tool which can customize agents through profiles[46]. By tweaking values for experience, curiosity, aggression, and other parameters, the agent will have a higher or lower priority to perform actions that are related to an objective with a certain characteristic. This originally aims at performing automated testing using characters in the C# game engine Unity.

Bibliography

[1] F. AI. *AIcrowd | NeurIPS 2021 - The NetHack Challenge | Challenges*. visited on 2023-1-16. June 2021. URL: https://www.aicrowd.com/challenges/neurips-2021-the-nethack-challenge#challenge-motivation.

[2] F. AI. *NetHack Challenge | NeurIPS 2021 NetHack Challenge website*. visited on 2023-10-19. June 2021. URL: https://nethackchallenge.com/report.html.

[3] P. Arcaini, A. Gargantini, and E. Riccobene. "Online testing of LTL properties for Java code". In: *Haifa Verification Conference*. Springer. 2013, pp. 95–111.

[4] S. Ariyurek, A. Betin-Can, and E. Surer. "Automated video game testing using synthetic and humanlike agents". In: *IEEE Transactions on Games* 13.1 (2019), pp. 50–67.

[5] L. Atkinson and M. Förderer. *gcovr – gcovr 6.0 documentation gcovr.com*. visited on 2023-9-19. 2013. URL: https://gcovr.com.

[6] A. Bauer, M. Leucker, and C. Schallhart. "Comparing LTL semantics for runtime verification". In: *Journal of Logic and Computation* 20.3 (2010), pp. 651–674.

[7] J. Bergdahl, C. Gordillo, K. Tollmar, and L. Gisslén. "Augmenting Automated Game Testing with Deep Reinforcement Learning". In: *CoRR* abs/2103.15819 (2021). arXiv: 2103.15819. URL: https://arxiv.org/abs/2103.15819.

[8] A. Botea, M. Müller, and J. Schaeffer. "Near optimal hierarchical path-finding." In: *J. Game Dev.* 1.1 (2004), pp. 1–30.

[9] M. Bratman. *Intention, Plans, and Practical Reason*. Cambridge: Cambridge, MA: Harvard University Press, 1987.

[10] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. *OpenAI Gym*. 2016. DOI: 10.48550/ARXIV.1606.01540. URL: https://arxiv.org/abs/1606.01540.

[11] C. Cifuentes, C. Hoermann, N. Keynes, L. Li, S. Long, E. Mealy, M. Mounteney, and B. Scholz. "BegBunch: benchmarking for C bug detection tools". In: *Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*. 2009, pp. 16–20.

[12] K. Claessen and J. Hughes. "QuickCheck: a lightweight tool for random testing of Haskell programs". In: *Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*. 2000, pp. 268–279.

[13] A. Elyasov, W. Prasetya, J. Hage, U. Rueda, T. Vos, and N. Condori-Fernández. "AB=BA: Execution Equivalence as a New Type of Testing Oracle". In: *Proceedings of the 30th Annual ACM Symposium on Applied Computing*. SAC '15. Salamanca, Spain: Association for Computing Machinery, 2015, pp. 1559–1566. ISBN: 9781450331968. DOI: 10.1145/2695664.2695877. URL: https://doi.org/10.1145/2695664.2695877.

[14] R. Ferdous, F. Kifetew, D. Prandi, I. S. W. B. Prasetya, S. Shirzadehhajimahmood, and A. Susi. "Search-Based Automated Play Testing of Computer Games: A Model-Based Approach". In: *Search-Based Software Engineering*. Ed. by U.-M. O'Reilly and X. Devroey. Cham: Springer International Publishing, 2021, pp. 56–71. ISBN: 978-3-030-88106-1.

[15] R. Ferdous, F. Kifetew, D. Prandi, and A. Susi. "Towards Agent-Based Testing of 3D Games Using Reinforcement Learning". In: *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 2022, pp. 1–8.

[16] G. Fraser and A. Arcuri. "EvoSuite: automatic test suite generation for object-oriented software". In: *ESEC/FSE '11*. 2011. URL: https://api.semanticscholar.org/CorpusID:10599913.

[17] D. Giannakopoulou and K. Havelund. "Automata-based verification of temporal properties on running programs". In: *Proceedings 16th Annual International Conference on Automated Software Engineering (ASE 2001)*. IEEE. 2001, pp. 412–416.

[18] C. Gordillo, J. Bergdahl, K. Tollmar, and L. Gisslén. "Improving playtesting coverage via curiosity driven reinforcement learning agents". In: *arXiv preprint arXiv:2103.13798* (2021).

[19] F. Hariri, A. Shi, and T. Lemberger. *A mutation tool for source and IR*. visited on 2023-9-19. Mar. 2018. URL: https://github.com/TestingResearchIllinois/srciror.

[20] S. Iftikhar, M. Z. Iqbal, M. U. Khan, and W. Mahmood. "An automated model based testing approach for platform games". In: *2015 ACM/IEEE 18th International Conference on Model Driven Engineering Languages and Systems (MODELS)*. 2015, pp. 426–435. DOI: 10.1109/MODELS.2015.7338274.

[21] Y. Jia and M. Harman. "An Analysis and Survey of the Development of Mutation Testing". In: *IEEE Transactions on Software Engineering* 37.5 (2011), pp. 649–678. DOI: 10.1109/TSE.2010.62.

[22] D. Jiménez. *Rydra/HierarchicalPathfinder: A C# implementation of a HPA* algorithm.* visited on 2023-1-27. Jan. 2017. URL: https://github.com/iv4xr-project/iv4xr-nethack.

[23] N. Justesen, P. Bontrager, J. Togelius, and S. Risi. "Deep learning for video game playing". In: *IEEE Transactions on Games* 12.1 (2019), pp. 1–20.

[24] J. K. *BotHack – A Nethack Bot Framework*. visited on 2023-9-19. Mar. 2014. URL: https://github.com/krajj7/BotHack.

[25] S. Karakovskiy and J. Togelius. "The Mario AI Benchmark and Competitions". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 55–67. DOI: 10.1109/TCIAIG.2012.2188528.

[26] P. P. Kumaresen, M. Frasheri, and E. Enoiu. "Agent-Based Software Testing: A Definition and Systematic Mapping Study". In: *CoRR* abs/2007.10224 (2020). arXiv: 2007.10224. URL: https://arxiv.org/abs/2007.10224.

[27] H. Küttler, N. Nardelli, A. H. Miller, R. Raileanu, M. Selvatici, E. Grefenstette, and T. Rocktäschel. *The NetHack Learning Environment*. 2020. DOI: 10.48550/ARXIV.2006.13760. URL: https://arxiv.org/abs/2006.13760.

[28] A. Lancichinetti, S. Fortunato, and F. Radicchi. "Benchmark graphs for testing community detection algorithms". In: *Physical review E* 78.4 (2008), p. 046110.

[29] A. Latos and S. C. Bakkes. "Automated Playtesting on 2D Video Games". In: (2022).

[30] Z. Li, Y. Wu, L. Ma, X. Xie, Y. Chen, and C. Fan. "GBGallery: A benchmark and framework for game testing". In: *Empirical Software Engineering* 27.6 (2022), pp. 1–27.

[31] G. Liu, M. Cai, L. Zhao, T. Qin, A. Brown, J. Bischoff, and T.-Y. Liu. "Inspector: Pixel-Based Automated Game Testing via Exploration, Detection, and Investigation". In: *2022 IEEE Conference on Games (CoG)*. IEEE. 2022, pp. 237–244.

[32] K. Lorber. *NetHack 3.6.6: NetHack Home Page*. visited on 2023-1-16. July 1999. URL: https://nethack.org.

[33] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. "Bugbench: Benchmarks for evaluating bug detection tools". In: *Workshop on the evaluation of software defect detection tools*. Vol. 5. Chicago, Illinois. 2005.

[34] F. Madeiral, S. Urli, M. Maia, and M. Monperrus. "Bears: An extensible java bug benchmark for automatic program repair studies". In: *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE. 2019, pp. 468–478.

[35] P. McMinn. "Search-based software testing: Past, present and future". In: *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE. 2011, pp. 153–163.

[36] N. Mu and J. Gilmer. "MNIST-C: A Robustness Benchmark for Computer Vision". In: (2019). DOI: 10.48550/ARXIV.1906.02337. URL: https://arxiv.org/abs/1906.02337.

[37] *NetHack, the amazing opensource Role-Playing Game from 1987 | INSAne*. visited on 2023-6-14. Oct. 2014. URL: http://insane.insa-rennes.fr/the-geek-corner-2/nethack-the-amazing-opensource-role-playing-game-from-1987/.

[38] L. O'Connor and O. Wickström. "Quickstrom: property-based acceptance testing with LTL specifications". In: *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 2022, pp. 1025–1038.

[39] C. Paduraru, M. Paduraru, and A. Stefanescu. "RiverGame - a game testing tool using artificial intelligence". In: *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. 2022, pp. 422–432. DOI: 10.1109/ICST53961.2022.00048.

[40] P. Patel and H. Hexmoor. "Designing BOTs with BDI agents". In: *2009 International Symposium on Collaborative Technologies and Systems*. IEEE. 2009, pp. 180–186.

[41] N. Pereira, P. Lima, E. Guerra, and P. Meirelles. "Towards Automated Playtesting in Game Development". In: *Anais Estendidos do XX Simpósio Brasileiro de Jogos e Entretenimento Digital*. Online: SBC, 2021, pp. 349–353. DOI: 10.5753/sbgames_estendido.2021.19666. URL: https://sol.sbc.org.br/index.php/sbgames_estendido/article/view/19666.

[42] I. Prasetya, M. Dastani, R. Prada, T. E. Vos, F. Dignum, and F. Kifetew. "Aplib: Tactical agents for testing computer games". In: *Engineering Multi-Agent Systems: 8th International Workshop, EMAS 2020, Auckland, New Zealand, May 8–9, 2020, Revised Selected Papers 8*. Springer. 2020, pp. 21–41.

[43] I. Prasetya, F. Pastor Ricós, F. M. Kifetew, D. Prandi, S. Shirzadehhajimahmood, T. E. Vos, P. Paska, K. Hovorka, R. Ferdous, A. Susi, et al. "An agent-based approach to automated game testing: an experience report". In: *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*. 2022, pp. 1–8.

[44] S. Shirzadehhajimahmood, I. Prasetya, F. Dignum, and M. Dastani. "An online agent-based search approach in automated computer game testing with model construction". In: *Proceedings of the 13th International Workshop on Automating Test Case Design, Selection and Evaluation*. 2022, pp. 45–52.

[45] L. Sonenberg. "Logics and collaboration". In: *Logic Journal of the IGPL* (May 2023). DOI: 10.1093/jigpal/jzad006.

[46] S. Stahlke, A. Nova, and P. Mirza-Babaei. "Artificial Players in the Design Process: Developing an Automated Testing Tool for Game Level and World Design". In: *Proceedings of the Annual Symposium on Computer-Human Interaction in Play*. CHI PLAY '20. Virtual Event, Canada: Association for Computing Machinery, 2020, pp. 267–280. ISBN: 9781450380744. DOI: 10.1145/3410404.3414249. URL: https://doi.org/10.1145/3410404.3414249.

[47] M. Utting, A. Pretschner, and B. Legeard. "A taxonomy of model-based testing approaches". In: *Software Testing, Verification and Reliability* 22.5 (2012), pp. 297–312. DOI: https://doi.org/10.1002/stvr.456. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.456. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.456.

[48] S. Varvaressos, K. Lavoie, A. Massé, S. Gaboury, and S. Hallé. "Automated Bug Finding in Video Games: A Case Study for Runtime Monitoring". In: Mar. 2014. DOI: 10.1109/ICST.2014.27.

[49] B. Wilkins and K. Stathis. "Learning to Identify Perceptual Bugs in 3D Video Games". In: *arXiv preprint arXiv:2202.12884* (2022).

[50] Y. Zheng, X. Xie, T. Su, L. Ma, J. Hao, Z. Meng, Y. Liu, R. Shen, Y. Chen, and C. Fan. "Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning". In: *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 2019, pp. 772–784. DOI: 10.1109/ASE.2019.00077.

Appendix A

Mutation testing results

Mutation killed	Line Number	Mutation
Change boolean operator		
Yes	20	if (obj->cursed ‖ !rn2(2)) {
Yes	28	} else if (!u.dx ‖ !u.dy) {
Invert conditions		
Yes	11	if (!(!getdir((char *) 0)))
Yes	14	if (!(obj->spe <= 0)) {
Yes	20	if (!(obj->cursed && !rn2(2))) {
Yes	22	} else if (!(u.uswallow)) {
Yes	25	} else if (!(u.dz)) {
Yes	28	} else if (!(!u.dx && !u.dy)) {
Yes	30	} else if (!((mtmp = bhit(u.dx, u.dy, COLNO, FLASHED_LIGHT,
Changed numeric boolean operators		
Yes	14	if (obj->spe != 0) {
Yes	14	if (obj->spe < 0) {
No	14	if (obj->spe == 0) {
Yes	14	if (obj->spe > 0) {
Yes	14	if (obj->spe >= 0) {
Yes	27	(u.dz != 0) ? surface(u.ux, u.uy) : ceiling(u.ux, u.uy));
Yes	27	(u.dz < 0) ? surface(u.ux, u.uy) : ceiling(u.ux, u.uy));
Yes	27	(u.dz <= 0) ? surface(u.ux, u.uy) : ceiling(u.ux, u.uy));
Yes	27	(u.dz == 0) ? surface(u.ux, u.uy) : ceiling(u.ux, u.uy));
No	27	(u.dz >= 0) ? surface(u.ux, u.uy) : ceiling(u.ux, u.uy));
Yes	32	(int FDECL((*), (OBJ_P, OBJ_P))) 0, &obj)) < 0) {
Yes	32	(int FDECL((*), (OBJ_P, OBJ_P))) 0, &obj)) <= 0) {
Yes	32	(int FDECL((*), (OBJ_P, OBJ_P))) 0, &obj)) == 0) {
No	32	(int FDECL((*), (OBJ_P, OBJ_P))) 0, &obj)) > 0) {
Yes	32	(int FDECL((*), (OBJ_P, OBJ_P))) 0, &obj)) >= 0) {

Mutation killed	Line Number	Mutation
Changed constants		
No	9	return -1;
No	9	return 1;
Yes	11	if (!getdir((char *) -1))
Yes	11	if (!getdir((char *) 1))
No	12	return -1;
No	12	return 1;
Yes	14	if (obj->spe <= -1) {
Yes	14	if (obj->spe <= 1) {
No	16	return -1;
No	16	return 0;
No	16	return 2;
No	20	if (obj->cursed && !rn2(-1)) {
No	20	if (obj->cursed && !rn2(-2)) {
No	20	if (obj->cursed && !rn2(0)) {
No	20	if (obj->cursed && !rn2(1)) {
No	20	if (obj->cursed && !rn2(3)) {
No	27	(u.dz > -1) ? surface(u.ux, u.uy) : ceiling(u.ux, u.uy));
Yes	27	(u.dz > 1) ? surface(u.ux, u.uy) : ceiling(u.ux, u.uy));
No	31	(int FDECL((*), (MONST_P, OBJ_P))) -1,
No	31	(int FDECL((*), (MONST_P, OBJ_P))) 1,
No	32	(int FDECL((*), (OBJ_P, OBJ_P))) -1, &obj)) != 0) {
No	32	(int FDECL((*), (OBJ_P, OBJ_P))) 1, &obj)) != 0) {
Yes	32	(int FDECL((*), (OBJ_P, OBJ_P))) 0, &obj)) != -1) {
Yes	32	(int FDECL((*), (OBJ_P, OBJ_P))) 0, &obj)) != 1) {
No	36	return -1;
No	36	return 0;
No	36	return 2;

Table A.1: *Full table of the mutation testing results of the camera mutation tests code shown in Listing 6.2. If the mutation was killed, it means one of the assertions in the testing code failed.* 56.9% *of mutations were killed during testing. The mutations that were not killed are mostly (*19.6%*) mutated return values, it checks whether a command was executed. The other mutations that were not killed are interactions with a cursed camera (*9.8%*) and the guard checking whether the camera flash hit another monster (*7.8%*), however this does not impact the game interaction.*

Appendix B

Nutrition tactic

```
public static Tactic resolveHungerState(int prayerTimeout) {
  // Tactic to resolve the hunger state. Consists out of 3 different methods.
  // Each method checks first whether the player is in a state where it wants food.
  return FIRSTof(
      // Prayer can resolve the hungers state by chance, since the gods can grant the player saturation.
      Actions.pray()
          .on(
              (AgentState S) -> {
                Player player = S.app().gameState.player;
                if (!player.hungerState.wantsFood()) {
                  return null;
                }
                // Logic to prevent praying again before the prayer timeout has passed.
                // If the player prays too frequently the gods will become angry
                Integer lastPrayerTurn = player.lastPrayerTurn;
                if (lastPrayerTurn == null
                    || lastPrayerTurn - S.app().gameState.stats.turn.time > prayerTimeout) {
                  return true;
                }
                return null;
              })
          .lift(),
      // If a corpse is still fresh, navigate to the corpse using the navigation tactic and eat it.
      NavTactic.interactWorldEntity(
          EntitySelector.freshCorpse.globalPredicate(
              S -> S.app().gameState.player.hungerState.wantsFood()),
          // 'y' and MORE command are used to confirm eating the corpse and to spend only one turn eating
          List.of(
              new Command(CommandEnum.COMMAND_EAT),
              new Command('y'),
              new Command(CommandEnum.MISC_MORE))),
      // Lastly an item from the inventory can be eaten in order to solve a state where the player is hungry
      Actions.eatItem()
          .on(
              (AgentState S) -> {
                Player player = S.app().gameState.player;
                // Player stomach is full enough
                if (!player.hungerState.wantsFood()) {
                  return null;
                }
                // Filter and sort the food items in the inventory
                List<Item> items =
                    Arrays.stream(player.inventory.items)
```

```
44                              .filter(
45                                  item ->
46                                      item instanceof FoodItem && ((FoodItem) item).foodInfo !=
                                            null)
47                              // Sort on nutrition per weight
48                              .sorted(
49                                  Comparator.comparingDouble(
50                                      item -> ((FoodItem) item).foodInfo.nutritionPerWeight))
51                              .toList();
52                      // No food items present in the inventory
53                      if (items.isEmpty()) {
54                        return null;
55                      }
56                      return items.get(0);
57                  })
58              .lift());
59      }
```

Listing B.1: *The code that defines the tactic for resolving the hunger state of the player. Each of the guards, defined in the 'on' contain checks to see if the player is hungry, this condition is in the 'globalPredicate' function for the action to eat a corpse. If the player is hungry and has not used a pray action in a long time, it will do so in order to resolve the hunger state without consuming any food. If that has not resolved the hunger state or the previous prayer time was too recent, the agent will try to make resolve the hunger state by trying to eat a fresh corpse. If this has not resolved the hunger state or there is no fresh corpse close by, it will try to select and eat food from its inventory. It sorts the food items on nutrition per weight, and it returns the item the player should eat. Then the .eatItem() action expects an item object it would eat. This tactic is part of the survival part of the agent tactic, because the agent could die from malnutrition.*