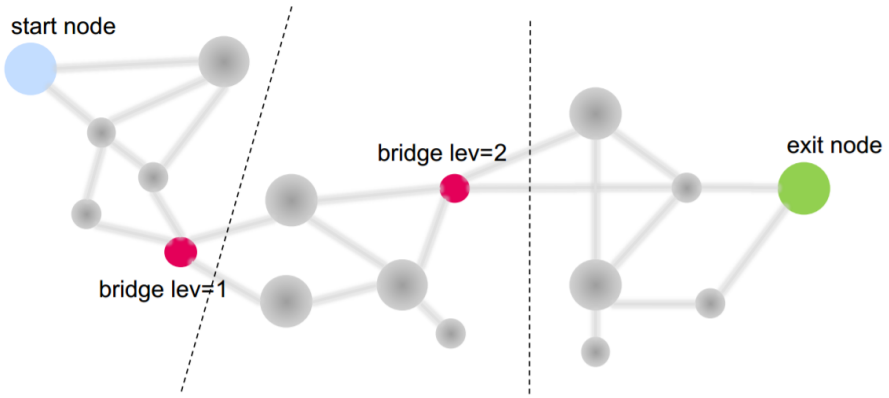
STV Project 2016/17, Iteration 2

Software Testing and Verification

W. van Steenbergen, J.Stolwijk and N.Hendrikx



Informatica

Universiteit Utrecht

Year 2016/2017

Contents

Introduction

Dear Reader,
This report covers our effort and contribution to the second iteration of the main project for Software testing and verification. Also it highlights the results and observations we had. The team members are Woudt van Steenbergen, Jesse Stolwijk and Niels Hendrikx.

Chapter 1

Finishing STV-rogue

This chapter will cover all the subjects of the mandatory part of this project of finishing STVRogue.

1.1 Opdracht 1.1 User Interface

Implemented a basic user interface giving the player information about their health and kills. The first menu is divided into a selection of options with a sub-menu where the player could choose a specific command to execute, this applies to the movement, use item, and attack command. When the location is contested the player will be able to see how many packs are on the node and have an option menu where they could select a pack to attack, this interface will display how many monsters are left in a pack and the overall health left on the pack. The player could also select the item menu and choose whether they want to use a potion or a crystal. Options that are not available given the constraints of using a crystal for example, will not be available until it is something the game allows the player to execute. This will eliminate the chance of the user causing exceptions to be thrown left, right, and center.

1.2 Opdracht 1.2 The game

All the game rules are applied in our project. Also we provided test with 100% block coverage. The test include the optional tests of the optional assignment 3.

Chapter 2

Testing Infrastructure

The Specification and GamePlay classes are located in utils/specification.cs
The tests for this part of the program are located in NTest_Replay
Solution root/Replay folder contains a "test suite" of replays

2.1 Runtime Instrumentation

game.ExecuteSpecification() hooks a instance of specification to the game.

2.2 Replay

GamePlay.serialize() save the gameplay (file XML format)
GamePlay.replay(Specification) replay the gameplay and hook a instance of specification.

2.3 More advanced types of specifications

Always
Unless
LeadsTo

2.4 Conditional Specifications

All 5 specification examples are implemented in NTest_Replay.

In the AgainstTest() a specification S will be tested if it holds for a replay suite and does not hold for the replay instance AgainstS.rpl.

Chapter 3

Testing STV-Rogues Game Rules

In the class NTest_GameRulesSpecs this optional assignment is implemented. The following game rules are tested with the specifications:

- RZone
- Rnode
- RAlert
- REndZone

We used the advised specifications from the project documentattion and for the EndZone rule we used a LeadsTo predicate. The optional tests provide 100% block coverage.

Chapter 4

Increasing Test Automation

Our approach to implementing AI Agents was restructuring the player class to make the player controlled agent an optional part of the game. This allowed agents that could be hooked up to the controls of the game. First we made some very basic agents before implementing the GoalAgent, which takes a collection of goals and attempts to reach those in sequence. The simple agents are meant to do one task, there is the PeaceFullRandomAgent, PrioritizeNewNodesAgent, and BloodThirstyAgent. The PeaceFullRandomAgent selects a random node from it's neighbours and moves to that one, until it reaches the exit. Then there's the PrioritizeNewNodesAgent, this is also a passive agent that selects a node from it's neighbours based on a precondition. If there is a node that has not yet been visited, the agent will prioritize these over ones already visited when moving around. In practice this caused the PrioritizeNewNodesAgent to reach the exit node anywhere between 10% and 200% faster than the PeaceFullRandomAgent with respect to times moved. This noticeable difference between performance comes from the PeaceFullRandomAgent being susceptible to chance. The last simple agent is the BloodThirstyAgent, this agent will choose to fight any packs it meets whenever the node becomes contested, when there's nothing to fight it will behave just like the PeaceFullRandomAgent. The GoalAgent bundles the features introduced with the simple agents and can execute them in sequence.

```
TupleList<string, int> goals;
goals = new TupleList<string, int> { {"reachup",3}, {"reachdown",1},
                {"kill",4}, {"exit",0} };
agent = new GoalAgent(game, randomseed, goals);
```

The features in GoalAgent are:

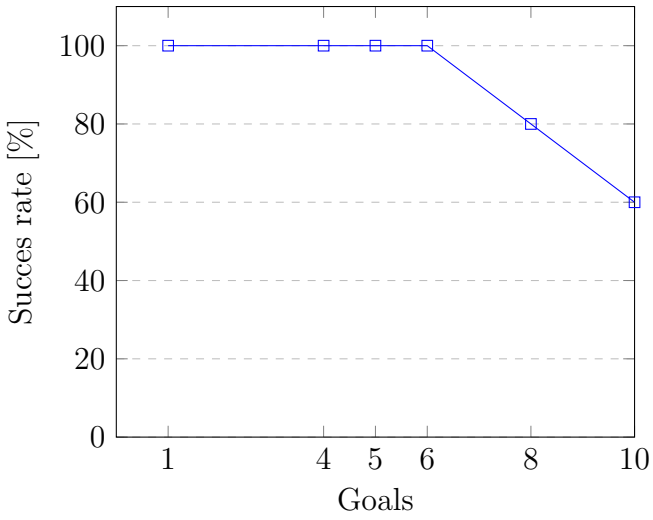
- ReachUp(int level): The agent will try to reach this zone at or above the player's current zonelevel.

- ReachDown(int level): The agent will backtrack over visited nodes to a node at or below the player's current zonelevel.
- Kill(int amount): The agent will roam around the dungeon and fight monsters until the amount is reached.
- Exit: The agent will act like PrioritizeNewNodesAgent and head straight for the exit. This should only be used as the last goal for the agent, since the game will terminate once the ExitNode is reached.

Whenever the agent reaches a terminal state: completing all goals, reaching the exit node as a result of the Exit goal, or arriving on the ExitNode prematurely. The agent will inform the user of how many goals it has managed to achieve before reaching a terminal state.

Assessing the performance of the GoalAgent empirically will be done through increasing the number of goals for the agent to reach, and observing how many times the agent manages to successfully complete all tasks before terminating. We expect the performance of the agent to go down the more goals it has to achieve. Each number of goals the agent will receive will be tested 5 times and graphed based on successful terminations. Each iteration a different seed will be used to diminish the effect of chance on the execution, but not completely rule out its effect. As can be seen in the graph 4 the performance of the agent worsens as it has to carry out more goals.

GoalAgent success rate dependent on number of goals



Chapter 5

Testing Statistics

5.1 Statistics

These statistics are covering every file in the project STVRogue. The unit test statistics are including the optional automated tests.

N = total # classes	37
locs = total # lines of codes()	794
locsavg = average # lines of codes()	locs/N = ˜21.5
Mavg = average # methods per class	215/37 = ˜5.8
Mmax = max # methods per class	11
cabe = the total mcCabe complexity	334
cabeavg = average mcCabe complexity per class	˜9.0

Table 5.1: General statistics

total # lines of codes of testing infrastructure	967
T = number of test cases ()	72
S = total number of tested specifications	5
Slocs = total # lines of codes of your specifications	T locs/N' = ˜78
E = total time spent on constructing specifications and their tests	65
Eavg = average effort	65/(72+5) = ˜0.84
bugs = total number of bugs ever found by this system-level testing	113

Table 5.2: System Test statistics