Universitat de Girona
Escola Politècnica Superior

Snowman Editor

A level design tool for the game A Good Snowman Is Hard To Build

Study: Grau en Enginyeria Informàtica

Document: Report

Author: Gerard Martin Teixidor

Email: gerardmartinteixidor@gmail.com

Director: Gustavo Ariel Patow and Mateu Villaret Auselle

Department: Informàtica, Matemàtica Aplicada i Estadística (IMAE)

Area: Llenguatges i Sistemes Informàtics (LSI)

Call: September/2018

Contents

Chapter 1

Introduction

In the design of video-games, one of the most important aspects to consider is the difficulty in the design of the different levels. A common problem is the possibility of finding possible solutions that were not planned. These solutions may not be desired because they can break the game dynamics. Hence, it is introduced the idea to create a tool to facilitate the design of those levels by detecting unwanted solutions.

This problem can be seen as a Planning problem, since, given the game state and the possible actions the player can perform, the solutions a level may have can be searched. Once the solutions are found, those that were not considered in the original design can be removed.

Usually this unwanted solutions are shorter, therefore easier solutions, than the designed solution. One of the goals of this project is, given a level, find the optimal solution, which will allow to check if there exist any solution easier than the designed one. Finding an optimal solution is not a simple task since exploring a game search tree by uninformed algorithms can be impossible in a reasonable time. Hence, one part of this project will be focused on reducing this previously described problem to Satisfiability Modulo Theories (*SMT*).

Since finding the optimal is a hard problem, in this project, two different problem relaxations will be presented which will allow to solve the problem in a plausible time.

1.1 Personal motivations

I had chosen this project because it merges the two worlds which I am more passionate about, video-games and optimization. Furthermore, is also a great way to go in depth about AI, since Planning is a well known branch of this topic.

This project has was not easy, despite this I enjoyed working on it and spending much free time on it.

1.2 Project objectives

The main objectives of this project is the development of this tool for a concrete game, *A Good Snowman is Hard to Build*. This tool, named *Snowman Editor*, will consist of the following parts:

- Solve a given level, with an heuristic approach, using the Planning Domain Definition Language (*PDDL*) and multiple solvers
- Solve a given level, with an optimal approach, using Satisfiability Modulo Theories (*SMT*)
- A graphical level editor

1.2.1 PDDL approach

The PDDL approach is an heuristic of solving multiple game instances. It can be split in to:

- Study the current *PDDL* language definition and its capabilities
- Search for the state of the art in *PDDL* planners
- Implement the problem with *PDDL*

1.2.2 SMT approach

The SMt approach is an optimal approach of solving multiple game instances. It can be split into:

- Study of *Planning as satisfiability*
- Study of the current *SMT-LIB* language capabilities
- Design an ad hoc *Planning as Satisfiability* like encoding for the game with *SMT*
- Implement the *SMT-LIB* encoding

1.2.3 Level editor

Graphical level editor to ease the creation of game instances and allow launching the two solver approaches. It can be split into:

- Layout design for the level editor
- Implement the level editor

The solving of different game instances will be called the *Snowman problem*.

Chapter 2

Feasibility study

The feasibility study can be split into technological and economical viability.

2.1 Technological viability

Since this project has a strong research component, it can not be ensured that all objectives will be accomplished. Specially, a realistic solving time was unknown at the start of this project, since the SMT solver is searching for optimality, which is a really hard problem.

2.2 Economical viability

2.2.1 Human costs

This analysis is based on the "a posteriori" results. The human costs is based on the following costs per hour:

- Programmer: 12 €/h
- Designer: 10 €/h
- Annalist: 18 €/h

Task	**Worker**	**Hours**	**Costs**
Study the current *PDDL*	Annalist	40	720
Search *PDDL* planners	Annalist	24	432
Implement problem with *PDDL*	Annalist	40	720
Study of *PAS*[1]	Annalist	24	432
Study of the current *SMT-LIB*	Annalist	8	144
Design *SMT-LIB* encoding	Annalist	320	5760
Implement *SMT* encoding	Programmer	160	2880
Layouts' design	Designer	8	96
Implement level editor	Programmer	80	960
Total			12144

The total human cost is 12144 €.

2.2.2 Equipment costs

The equipment cost takes into account the hardware and software licences used in this project.

[1] Planning as satisfiability

Item	**Price**
Developer PC	1508
Compunting PC	1290
A Good Snowman Is Hard To Build	10
Java Development Kit 8	0
Yices 2	0
IntelliJ IDEA Community 2018.1	0
Debian 9	0
Ubuntu 18.04	0
Gimp 2.8	0
Sublime Text 3	80
Trello	0
GitHub	87
BitBucket	0
Lucidchart	5
Total	2970

This project used two computers, one for developing and the other for computing the results, because this can take several hours.

2.2.3 Total costs

In conclusion, the estimated total cost of the development of the project is 124410 €.

Chapter 3

Methodology

The methodology used by this project is an Agile-like development. Because this project has a significant research component, it was needed an iterative and incremental evolution. Each iteration of the methodology has the following steps:

- Problem analysis and description
- Possible theoretical problem solution
- Toy problem implementation to prove the solution
- Toy problem implementation testing
- Real problem implementation
- Real problem implementation testing

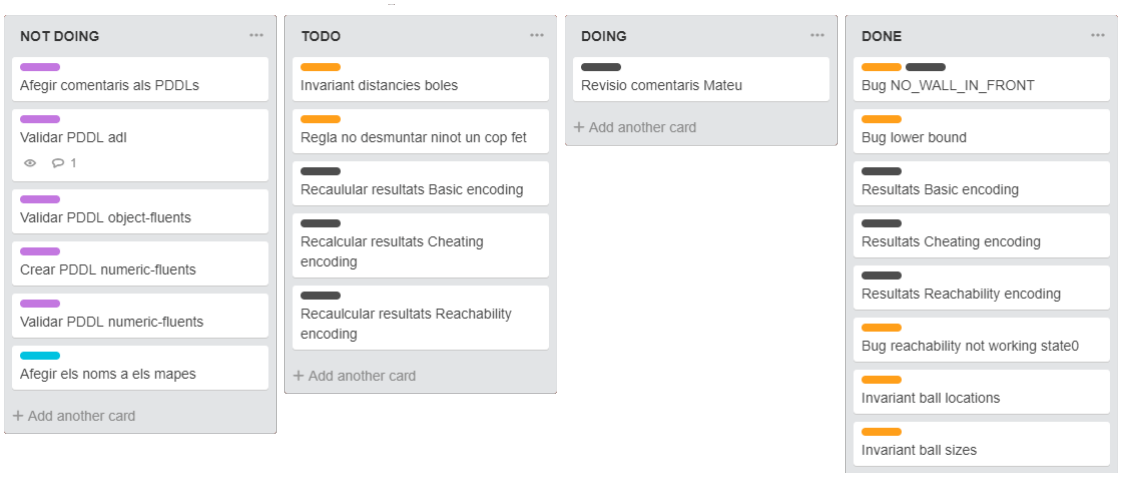
During the implementation, multiple optimizations were found and marked to be implemented in the next project iteration. The main project iterations are:

- PDDL *adl* approach
- PDDL *object-fluents* approach
- SMT *Basic encoding*
- SMT *Cheating encoding*
- SMT *Reachability encoding*

Also, an agile board has been used to keep track of the multiple tasks.

3.1 Agile board

An agile board is a board which gives a visual representation of the project tasks and its current status. Usually the board contains a column for each status which can be *To do*, *Doing*, *Done*, *Backlog*, etc.



Chapter 4

Project planning

This chapter describes all the details of the project planning which has been made before the project work.

4.1 Working plan

The solver related tasks will follow the Agile methodology because these tasks are related to research and have unexpected results while the level editor will be deployed in a single iteration at the end of the project, once it have all the requirements from the solvers.

4.2 Planned tasks

4.2.1 Study the current *PDDL*

Study the different PDDL versions and its expressive capabilities. Search for implementation examples and good practices.

4.2.2 Search *PDDL* planners

Search for publicly available PDDL planners and determine which subset of the language they support. The main source of planners are the *International Planning Competitions*. Also, perform some benchmarks to find which one to use in the *Snowman problem*.

4.2.3 Implement problem with *PDDL*

Implement the *Snowman problem* with PDDL. Create the domain and implement a level to PDDL instance algorithm.

4.2.4 Study of *Planning as satisfiability*

Analysis and study of the planning as satisfiability techniques and encodings. This will be our most successfull approach used to solve the *Snowman problem*.

4.2.5 Study of the current *SMT-LIB*

Analysis and study of the current implementation of *SMT-LIB 2.6* and its expressiveness. For this project it is needed the *Ints* theory.

4.2.6 Design *SMT-LIB* encoding

Design an SMT encoding using the *SMT-LIB* language for the *Snowman problem*.

4.2.7 Implement *SMT-LIB* encoding

Implement an application which given a *Snowman problem* instance can generate its corresponding *SMT-LIB* encoding.

4.2.8 Layouts' design

Design the corresponding GUI layouts which will be used in the level editor.

4.2.9 Implement level editor

Implement a graphical level editor which also is an interface for all the previous functionalities.

4.2.10 Documentation

Write all the documentation, including this document.

4.3 Estimated scheduling

		Feb 2017				Mar 2017				Apr 2017			
Task Name	**Duration**	**1W**	**2W**	**3W**	**4W**	**1W**	**2W**	**3W**	**4W**	**1W**	**2W**	**3W**	**4W**
Study the current *PDDL*	1												
Search *PDDL* planners	1												
Implement problem with *PDDL*	1												
Study of *PAS*[1]	1												
Study of the current *SMT-LIB*	1												
Design *SMT-LIB* encoding	8												
Implement *SMT-LIB* encoding	4												
Layouts' design	1												
Implement level editor	2												
Documentation	4												

	May 2017				Jun 2017				Jul 2017				Aug 2017			
Task Name	**1W**	**2W**	**3W**	**4W**	**1W**	**2W**	**3W**	**4W**	**1W**	**2W**	**3W**	**4W**	**1W**	**2W**	**3W**	**4W**
Study the current *PDDL*																
Search *PDDL* planners																
Implement problem with *PDDL*																
Study of *PAS*																
Study of the current *SMT-LIB*																
Design *SMT-LIB*																
Implement *SMT-LIB* encoding																
Layouts' design																
Implement level editor																
Documentation																

[2]

[1] Planning as satisfiability

[2] This table is an approximation and does not express the iteration process that will be done during the five first months.

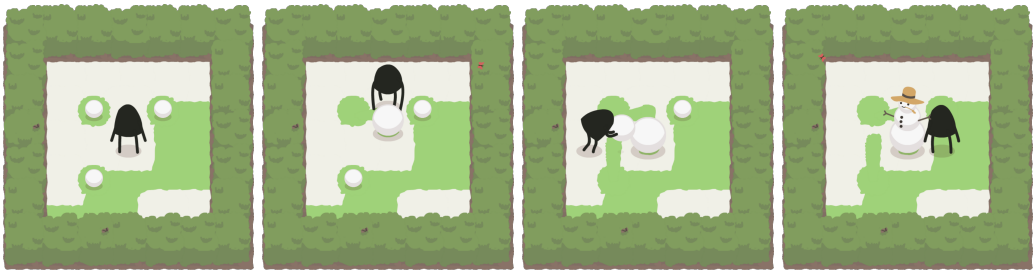
Chapter 5

Framework

This chapter describes all the needed background to understand the project.

5.1 A Good Snowman is Hard to Build

A Good Snowman is Hard to Build is a commercial game based on the popular Sokoban transport puzzle. The goal is to build a snowman from tree balls, each one smaller than the other.



The game consists of multiple levels arranged in an open world layout. For this project each individual level will be seen as a different problem instance. A complete instance solution can be found in the Example 5.1.1.

5.1.1 Game mechanics

The game has two worlds with different mechanics each one. This project is focused on the "real world" mechanics but, with minor changes, it should be easy to adapt it to the "dream world" mechanics. The rules describing the "real world" mechanics are the following:

1. The character can only push the balls from behind.
2. The character can also push and pop a ball over other balls.
3. A ball can not be pop from a ball to directly be on top of another ball.
4. The ball below must be larger.
5. Each ball can have three sizes: small, medium and large.
6. When the ball is rolled on top of snow on the floor it increases its size by one unit, and removes the snow on the floor.
7. When rolled, a large ball does not increase its size though it also removes the snow.
8. Balls can only grow.
9. A level can have multiple snowmans. The number of balls in a level has to be a multiple of three and bigger than 0.
10. Once the snowman is built, you can not pop the top ball. This rule currently is not implemented in the SMT approach.
11. A snowman is formed by three balls.
12. A snowman can be built anywhere.

Example 5.1.1. *A complete solution to the Andy level:*

5.2 Automated planning

Automated planning or AI Planning is a branch of artificial intelligence which aims at finding sequences of actions to achieve certain given goals from some initial conditions. It has been an area of research in artificial intelligence for over three decades. Planning techniques have been applied in a variety of tasks including robotics, process planning, web-based information gathering, autonomous agents and spacecraft mission control. Planning involves the representation of actions and world models, reasoning about the effects of actions, and techniques for efficiently searching the space of possible plans.

A planning problems is typically described by its initial state, a description of the desired goal and a description of possible actions which, given a state, generate other states. A solution to a planning, or plan, is a set of action, when applied to any of the initial states, it generates a goal state. Finding such a goal state turns to be a searching problem in a graph space, for instance Figure 5.2.

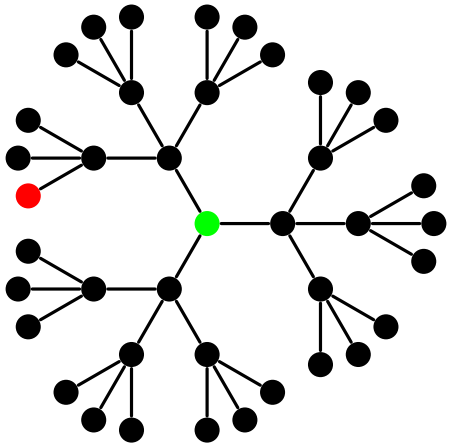


Figure 5.1: All possible states from a planning problem with 3 actions applied 3 times from the initial state (green). Possible solution marked as red.

A typical example is the classical *blocks world* domain where there are three blocks (A, B and C) that can be piled on a table. The possible actions are picking blocks and letting blocks on top of a block or on the table. These actions have preconditions forbidding their executions (for instance a block that is under another block can not be picked) and effects that change the state of the world. A particular instance of the problem consists on a configuration of the blocks, for example, A is on top of B which is on top of C which is on the table, and a desired goal state, for example, C is on top of B which is on top of A which is on the table. For this particular problem a possible plan would be these consecutive actions: pick A and let it on the table, pick B and let it on A and pick C and let it on B.

5.3 Planning Domain Definition Language

The Planning Domain Definition Language (PDDL) [1, 2, 3, 4, 5, 6] is a standard Artificial Intelligence planning language. It's the official language for the International Planning Competition (IPC) [7], which has evolved with each competition. The current PDDL version is the 3.1 [6].

The language separates the domain description and the (particular instance) problem description. The domain defines all possible actions from a plan, while the problem defines the initial state and possible goal states. An example can be seen in Example 5.3.1.

Example 5.3.1. *Problem example where a robot has to move the ball 1 from room A to room B. The problem contains the following actions:*

move (?from ?to) *move the robot from the room ?from to the room ?to*

pick (?obj ?room ?gripper) *pick an object ?obj with the arm ?gripper which is in the room ?room*

drop (?obj ?room ?gripper) *drop an object ?obj from the arm ?gripper to the room ?room*

The problem uses the following predicates:

(room ?r) *true if the object ?b is a room*

(ball ?b) *true if the object ?b is a ball*

(gripper ?g) *true if the object ?b is an arm*

(at-robby ?r) *true if the robot is in the room ?r*

(at ?b ?r) *true if the object ?b is in the room ?r*

(free ?g) *true if the ram ?g is not carrying anything*

(carry ?o ?g) *true if the object ?o is been carried by the arm ?g*

Note that the action parameters can be instantiated by any object, for example in the action move, the parameter ?from can be instantiated the object rooma but also can be the object left. To restrict only the rooms to be valid, the predicate (room ?r) is needed.

The domain description:

```
(define (domain gripper-strips)
   (:predicates (room ?r)
        (ball ?b)
        (gripper ?g)
        (at-robby ?r)
        (at ?b ?r)
        (free ?g)
        (carry ?o ?g))

   (:action move
       :parameters (?from ?to)
       :precondition (and (room ?from) (room ?to) (at-robby ?from))
       :effect (and (at-robby ?to)
            (not (at-robby ?from))))

   (:action pick
       :parameters (?obj ?room ?gripper)
       :precondition (and (ball ?obj) (room ?room) (gripper ?gripper)
                (at ?obj ?room) (at-robby ?room) (free ?gripper))
       :effect (and (carry ?obj ?gripper)
            (not (at ?obj ?room))
            (not (free ?gripper))))

   (:action drop
       :parameters (?obj ?room ?gripper)
       :precondition (and (ball ?obj) (room ?room) (gripper ?gripper)
```

```
            (carry ?obj ?gripper) (at-robby ?room))
    :effect (and (at ?obj ?room)
         (free ?gripper)
         (not (carry ?obj ?gripper)))))
```

Predicates are initialized inside the block init, i.e., (:init (room rooma)) initializes the predicate room with the object rooma to true.

The problem definition with the initial state and the goal restrictions:

```
(define (problem strips-gripper2)
    (:domain gripper-strips)
    (:objects rooma roomb ball1 ball2 left right)
    (:init (room rooma)
           (room roomb)
           (ball ball1)
           (ball ball2)
           (gripper left)
           (gripper right)
           (at-robby rooma)
           (free left)
           (free right)
           (at ball1 rooma)
           (at ball2 rooma))
    (:goal (at ball1 roomb)))
```

5.3.1 Subset adl

The *adl* subset is one of the most basic PDDL language subsets using:

- strips: STRIPS language-like planning problem definition (see [8] for further STRIPS details)
- typing: Group objects into types
- negative-preconditions: Use of *not* predicate in preconditions
- equality: Equality between objects
- quantified-preconditions: Use of *forall* and *exists* in preconditions. The language uses a closed-world assumption
- conditional-effects: Use of *when* predicate in effects

Since any problem can be described only using *strips*, *adl* facilitates the language syntax.

5.3.2 Subset numeric-fluents

The *numeric-fluents* subset allows a new kind of predicates named *function* which can return an integer number.

5.3.3 Subset object-fluents

The *object-fluents* subset is the latest language subset which allows the functions to return objects instead of the integer number type. This allows the language to be more expressive.

5.4 Planner

A planner is a domain-independent software which aims to solve a planning problem. Usually the input is a PDDL-model, composed by the domain and the problem.

5.5 Boolean Formulas

Given a set `X` of Boolean variables, a Boolean formula `F` can be constructed by the grammar:

```
F ::= X
   |  not F
   |  F \/ F
   |  F /\ F
   |  F -> F
   | ( F )
```

where disjunction ($\vee$) and conjunction ($\wedge$) operators are associative. Further operators exist but do not add further complexity. Moreover, all formulas can be equivalently expressed using just the negation ($\neg$), disjunction and conjunction operators since, in fact, $A \rightarrow B$ is equivalent to $\neg A \vee B$.

5.6 Propositional satisfiability problem

An interpretation is an assignment (or mapping) of truth values (true or false) to Boolean variables.

The propositional satisfiability problem (SAT) is the well known problem of determining, if given a Boolean formula, there exists any interpretation that satisfies it. In other words, if all variables can be replaced by their assigned truth values and the formula evaluates to *true*.

Example 5.6.1. *An example of a SAT problem.*

$$\begin{aligned}&(\neg A \vee \neg B \vee \neg C) \quad \wedge \\ &(A \vee \neg C) \quad \wedge \\ &(\neg B \vee E \vee \neg E) \quad \wedge \\ &(C)\end{aligned}$$

A possible interpretation satisfying the formula could be $\{A, \neg B, C, D, E\}$.

SAT has been proven to be NP-complete, which means that all problems in the class NP are at most as difficult to solve as SAT.

5.7 SAT Modulo Theories

The satisfiability modulo theories (SMT) problem is a decision problem for logical formulas with respect to a background theory, for example the theory of integers. It can be though as a form of SAT where instead of having just Boolean variables or their negations as literals, it can also have atoms that will be evaluated to true or false according to some background theories, e.g., $(x < 2 + y)$, where the predicate $<$ is comparing two arithmetic expressions containing integer variables x and y. See [9] for further details.

The SMT-LIB is a library and set of standards for SMT systems, which includes a language. The latest language version is the *2.6*.

Possible *SMT-LIB 2.0* Logical operators are:

- main logical operations: (and), (or), (xor), (not)
- implication (=>), equality (=) and distinction (distinct)
- it-then-else (ite)

In addition, the *Ints* theory allow the following operations:

- addition (+) and substraction (-)
- multiplication (*), division (div) and modulo (mod)
- absolute value (abs)
- smaller than (<), smaller or equal than (<=), larger than (>), larger or equal than (>=)

Example 5.7.1. *An SMT problem using* SMT-LIB 2.0 *and* Ints *theory with Linear Integer Arithmetic (LIA) logics:*

```
(set-logic QF_LIA)
(declare-const x Int)
(declare-const y Int)
(declare-const a Bool)
(assert (= a true))
(assert (and (= (- x y) (+ x (- y) 1)) a))
(check-sat)
(exit)
```

The result is unsat, meaning there isn't an interpretation, or a possible assignment which satisfies the conjunction of the formulas:

$$a \leftrightarrow true \tag{5.1}$$

$$x - y = (x + (-y) + 1) \quad \wedge \quad a \tag{5.2}$$

5.8 Planning as Satisfiability

Planning as Satisfiability is a planning resolution technique based in reducing the problem to SAT by encoding as a Boolean formula the question "is there a sequence of n actions that brings from state S^0 to a final state?". In other words, it consists in, for a given number of steps n, defining $n+1$ possible states (being S^0 the initial state and S^n the final one) and choosing n actions to perform these transitions from state to state. The initial state variables are initialized by the encoding using the instance, meanwhile the other states variables will be resolved by the SAT solver.

The encoding must ensure that the chosen action at time step t can be performed, since its precondition is met at state S^t and it must also ensure that their effects are applied at state S^{t+1}. Moreover, the encoding has also to ensure that only one action is chosen and that variables not involved in the effect of the chosen action do not "freely" change from one state to the other.

It is important to highlight that an encoding of n time steps encodes a planning problem of exactly n actions. To check if there exists a plan for n or fewer actions, it is needed to generate and solve n different encodings, each one with a one more time step than the previous one. Figure 5.2 illustrates the formula for n time-steps.

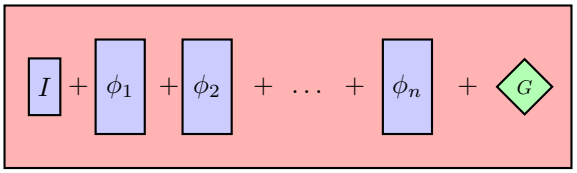


Figure 5.2: SAT formula encoding plan feasibility if n time-steps. I are S^0 assignments, ϕ_is are formulas encoding sound transition between states according to a chosen action for the i-th time-step and G are the goal state restrictions.

When the problem to model requires, for instance, integer variables to represent the state, it makes sense to consider planning as satisfiability modulo theories instead of just planning as satisfiability (see for instance [10]).

Chapter 6

System requirements

The system requirements can be split into functional and nonfunctional requirements.

6.1 Functional requirements

The Snowman editor functional requirements are the followings:

- The user must be able to create a custom level
- The user must be able to play the level
- The user must be able to save the level
- The user must be able to load a level
- The user must be able get the optimal solution to a level
- The user must be able to get an approximate solution to a level

6.2 Nonfunctional requirements

To run the Snowman Editor the following requirements must be satisfied:

6.2.0.1 Hardware requirements

OS: Any *Linux* distribution (64 bits). *Windows* [1] and *Mac OS X* are also supported but not tested.

CPU: Any. It is recommended to have a good single thread performance.

RAM: 16 GB. It is recommended to have high speed and low latency ram.

6.2.0.2 Software requirements

- *Yices 2*. Tested with version 2.5.4
- *A Good Snowman Is Hard To Build*. Tested with the Steam content BuildID 1086215
- *Java SE Runtime Environment 8*
- *Java Development Kit 8*

[1] Windows seems to have performance issues with *Yices 2*

Chapter 7

Studies and decisions

This chapter describes all the studies and decisions that have been made during the development of this project.

7.1 PDDL as a problem description language

Instead of performing an ad hoc approach to the *Snowman problem*, an alternative is to use existing software that solves generic planning problems by defining the problem as an input. The most used and updated planning description language is PDDL.

7.2 Scala as a programming language

Scala has been chosen to be the main developing language because, like Java, it is an operative system independent language. Also its functional approach allows for a more clear, simple to write, code.

7.3 Functional programming

Most of this project's code follows the functional program paradigm. There are some cases where imperative programming was added to improve performance or to simplify the code structure. Since this project does not have a big parallelism component, the functional programming is not mandatory and the main reason it has been chosen is to explore this paradigm.

7.4 SMT versus SAT

To facilitate the encoding programming, the problem is encoded in SMT instead of SAT. This allows the use of Integer theory and avoids its codification to SAT. Anyway, a translation from SMT to SAT is possible and should be as simple as implementing a SAT *translator*.

7.5 Yices 2 as an SMT Solver

Yices 2[11, 12] is an open source *SMT* solver which can process input written in the *SMT-LIB* notation. It has been chosen because, according to the knowledge of the group where I'm developing the project, it is one of the best solvers that supports *SMT-LIB 2.0*. However, it should not be difficult to use any other SMT solver also supporting *SMT-LIB 2.0*.

Chapter 8

Analysis and Design

This chapter describes the all work done and its details.

8.1 Snowman Editor

The *Snowman Editor* is a software package which includes all the work done in this project. This software allows to edit, play and solve *Snowman problem* instances. The software code is split in four modules each one encapsulating a different functionality.

- Planner module
- Snowman problem module
- Game modification module
- GUI module

Because the code contains more than a hundred classes, without including the companion objects for some of these classes, the following analysis only focuses on the more interesting parts.

8.1.1 Planner module

The *planner* module (package *planner*) is an abstraction of a planning as satisfiability solver. By giving a custom *encoder*, *translator* and *solver*, it will search for a plan. The pipeline can be seen in Figure 8.1. A complete class diagram can be seen in Figure 8.8.

The *planner* uses an abstract *internal language* similar to *SMT-LIB*. The parse tree (Figure 8.3) is represented by classes subclassing, but because of the Scala language limitations, a more generic parsing tree (Figure 8.2) has been used, but checking the types during run-time.

Example 8.1.1. *Scala code examples as an abstraction of the* Internal language*:*

```
And(Equals(b.x, bNext.x), Equals(b.y, bNext.y), Equals(b.size, bNext.size))
Smaller(stateNext.mediumBalls, IntegerConstant(2 * level.snowmans + 1))
Equivalent(eff, actionVariable)
```

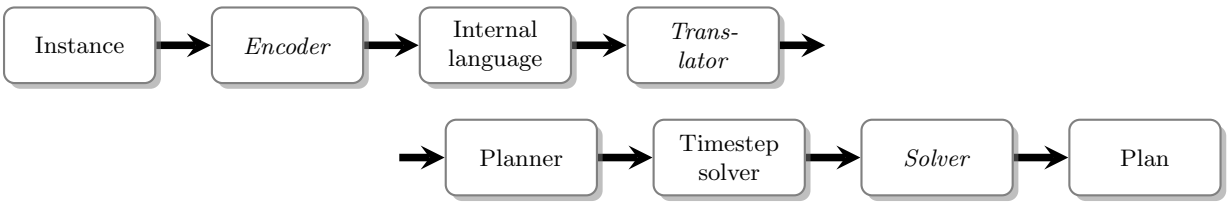


Figure 8.1: The planner pipeline

```
Expressison -> expression (ClauseDeclaration | VariavleDeclaration | Comment)
ClauseDeclaration -> clauseDeclaration Clause
VariavleDeclaration -> variableDeclaration Clause
Comment -> comment [Scala String]
Clause -> clause (Add | Or | Not | Implies | Equivalent | BooleanVariable |
    BooleanConstant | Add | Sub | Equals | Smaller | Greater |
    IntegerVariable | IntegerConstant)
And -> and Clause Clause
Or -> or Clause Clause
Not -> not Clause
Implies -> implies Clause Clause
Equivalent -> equivalent Clause Clause
BooleanVariable -> booleanVariable [Scala Boolean]
BooleanConstant -> booleanConstant [Scala Boolean]
Add -> add Clause Clause
Sub -> sub Clause Clause
Equal -> equal Clause Clause
Smaller -> smaller Clause Clause
Greater -> greater Clause Clause
IntegerVariable -> integerVariable [Scala Int]
IntegerConstant -> integerConstant [Scala Int]
```

Figure 8.2: *Internal language* generic parse tree

```
Expressison -> expression (ClauseDeclaration | VariavleDeclaration | Comment)
ClauseDeclaration -> clauseDeclaration (BooleanClause | IntegerClause)
VariavleDeclaration ->  variableDeclaration (BooleanVariable | IntegerVariable)
Comment -> comment [Scala String]
BooleanClause -> booleanClause (Add | Or | Not | Implies | Equivalent | Smaller |
    Greater | Equals)
IntegerClause -> integerClause (Add | Sub | IntegerVariable | IntegerConstant)
And -> and BooleanClause BooleanClause
Or -> or BooleanClause BooleanClause
Not -> not BooleanClause
Implies -> implies BooleanClause BooleanClause
Equivalent -> equivalent BooleanClause BooleanClause
BooleanVariable -> booleanVariable [Scala Boolean]
BooleanConstant -> booleanConstant [Scala Boolean]
Add -> add IntegerClause IntegerClause
Sub -> sub IntegerClause IntegerClause
Equal -> equal IntegerClause IntegerClause
Smaller -> smaller IntegerClause IntegerClause
Greater -> greater IntegerClause IntegerClause
IntegerVariable -> integerVariable [Scala Int]
IntegerConstant -> integerConstant [Scala Int]
```

Figure 8.3: *Internal language* parse tree

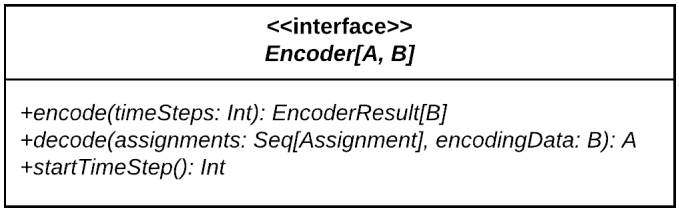


Figure 8.4: Encoder interface

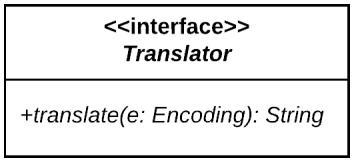


Figure 8.5: Translator interface

8.1.1.1 Encoder

The *encoder* is an abstraction (Figure 8.4) which defines encoding of the problem instance to the *internal language*. Also is responsible to translate the *solver* result to *planner* actions.

8.1.1.2 Translator

The *translator* is an abstraction (Figure 8.5) which defines the translation of the the internal language to the *solver* language. The solver language is a generic type which is defined by the solver.

8.1.1.3 Solver

The *solver* an abstraction (Figure 8.6) which defines the interface between the planner and the external solver. It inputs the *translator* result to the solver and passes the output to the *encoder* to decode.

8.1.1.4 Planner submodule and Timestep solver

The *Planner submodule* is the main solver sub-module which launches multiple *Timestep solvers* in parallel or sequentially. Meanwhile, the *Timestep solver* is a sub-module which solves an encoding of n time steps. The planner will launch different *Timestep solvers*, with an increasing formula of n time steps, until a solution is found.

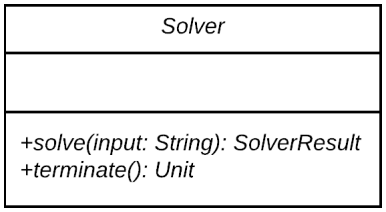


Figure 8.6: Solver abstract class

Cores	Time (s)	Ratio
1	345754	1
2	201991	1.712
3	165163	2.083
4	168111	2.057
5	181119	1.909
6	199880	1.730
7	203140	1.702
8	224669	1.539

Figure 8.7: Time results solving the Sarah level using the *Basic encoding*

The planner is able to use parallelism by launching multiple *Timestep solvers* at the same time, each one with an encoding for a different number of time steps. If a solution is found in n time steps, it will kill all the executions with a bigger time step and will finish the lower time step encoding already running. Once all of them are finished, will pick the solution with the lower time step as the final solution.

Using a single machine (with 4 cores and hyperthreading), the best performance seems to be using 3 threads (Figure 8.7), this can be caused by the cache sharing since the SAT/SMT solvers usually are memory dependent.

8.1.1.5 Packages

Description of the packages found in this module.

encoder Encoder abstraction interfices. A complete class diagram can be seen in Figure 8.10

operations *Internal language* parse tree definition, run-time checks and utilities

planner Main functions for the planner

solver Solver abstraction interfices

timestep Single formula encoding solver

translator Translator interfices

8.1.2 Snowman problem module

The *snowman problem* module (package *snowman*) is the concrete implementation of the planning problem. It implements all the abstractions needed by the *planner*. Also it is responsible of generation the *PDDL problem* given an instance.

Also this module includes all the data structures to represent a game level and its properties.

The module contains one *encoder* for each of the three encodings. To avoid code repetition between the three encoders, a generic Template (figure 8.9) pattern has been used in the encoder classes, while a Strategy pattern has been used with the states attributes. Also, the encoders are instantiated using a Factory in the *BaseEncoder* object.

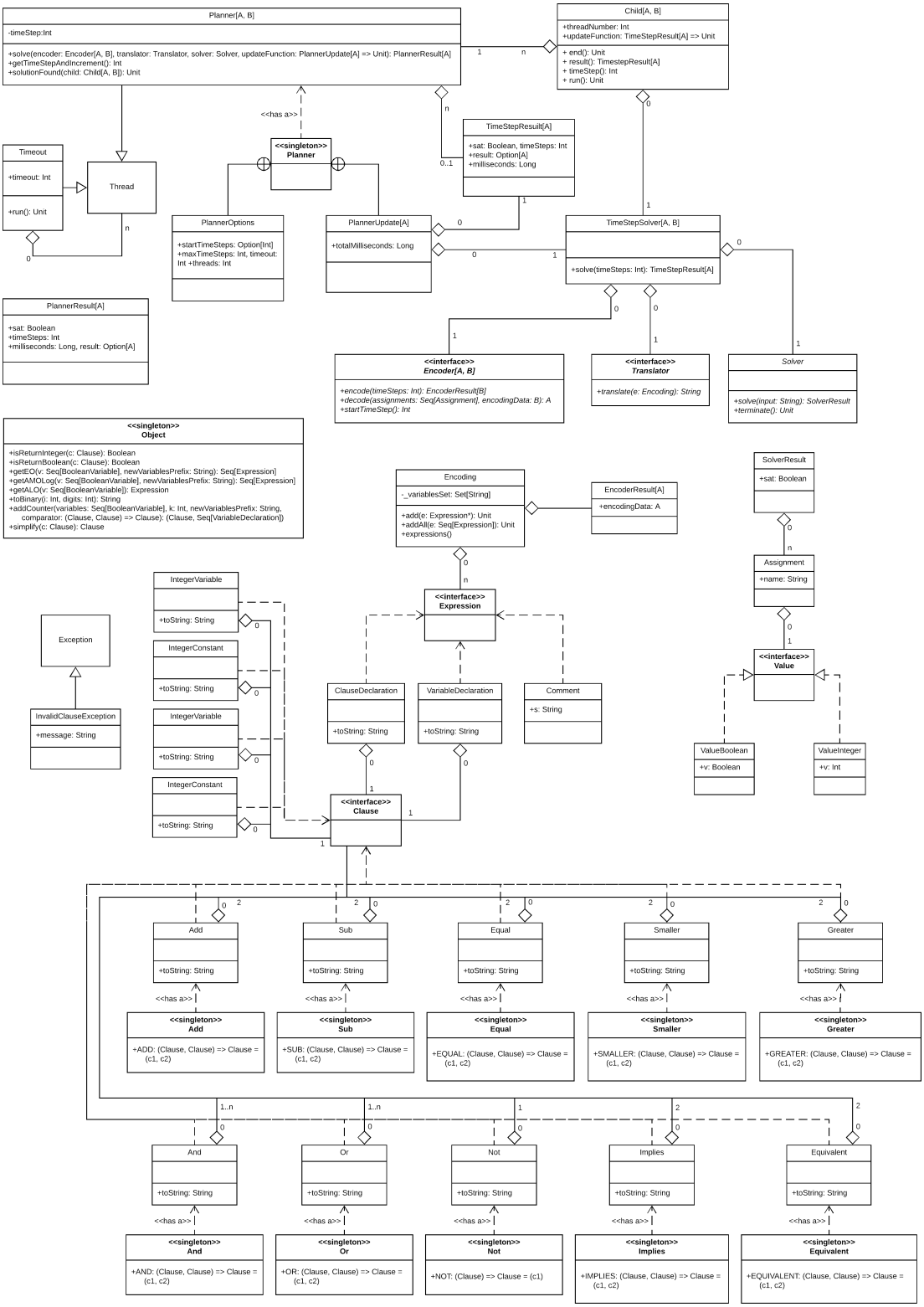


Figure 8.8: Class diagram from the planner module

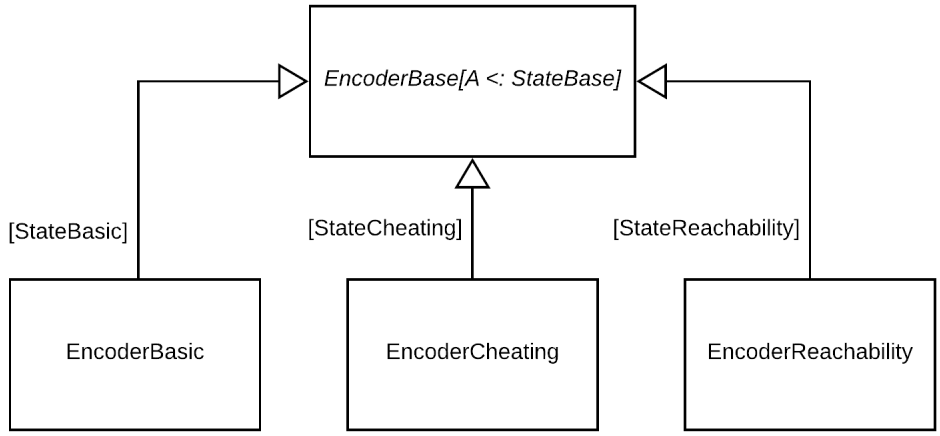


Figure 8.9: Generic template class diagram

8.1.2.1 Packages

Description of the pacakges found in this module.

action Actions the snowman planning problem can perform

collection Custom collections

encoder Concrete *planner* encoder implementation. Contains all the encodings

game Game constants and description

level Level data structure

pddl PDDL instances generator

planner Submodule which interfaces with the planner module

solver Concrete *planner* solver implementation

translator Concrete *planner* translator implementation

util Various utilities

validator Planning validator

8.1.3 Game modification module

The *game modification* module (package *mod*) is in charge of altering the game files to launch the game with a custom level. Also, it allows to restore the game to the previous state. A custom levels is being loaded by altering multiple resources files and the save game files.

8.1.4 GUI module

Finally, the *GUI* module (package *ui*) is the level editor, but also the main module which calls all the other modules. The *GUI* is written in a native way using *Scala/Java Swing*.

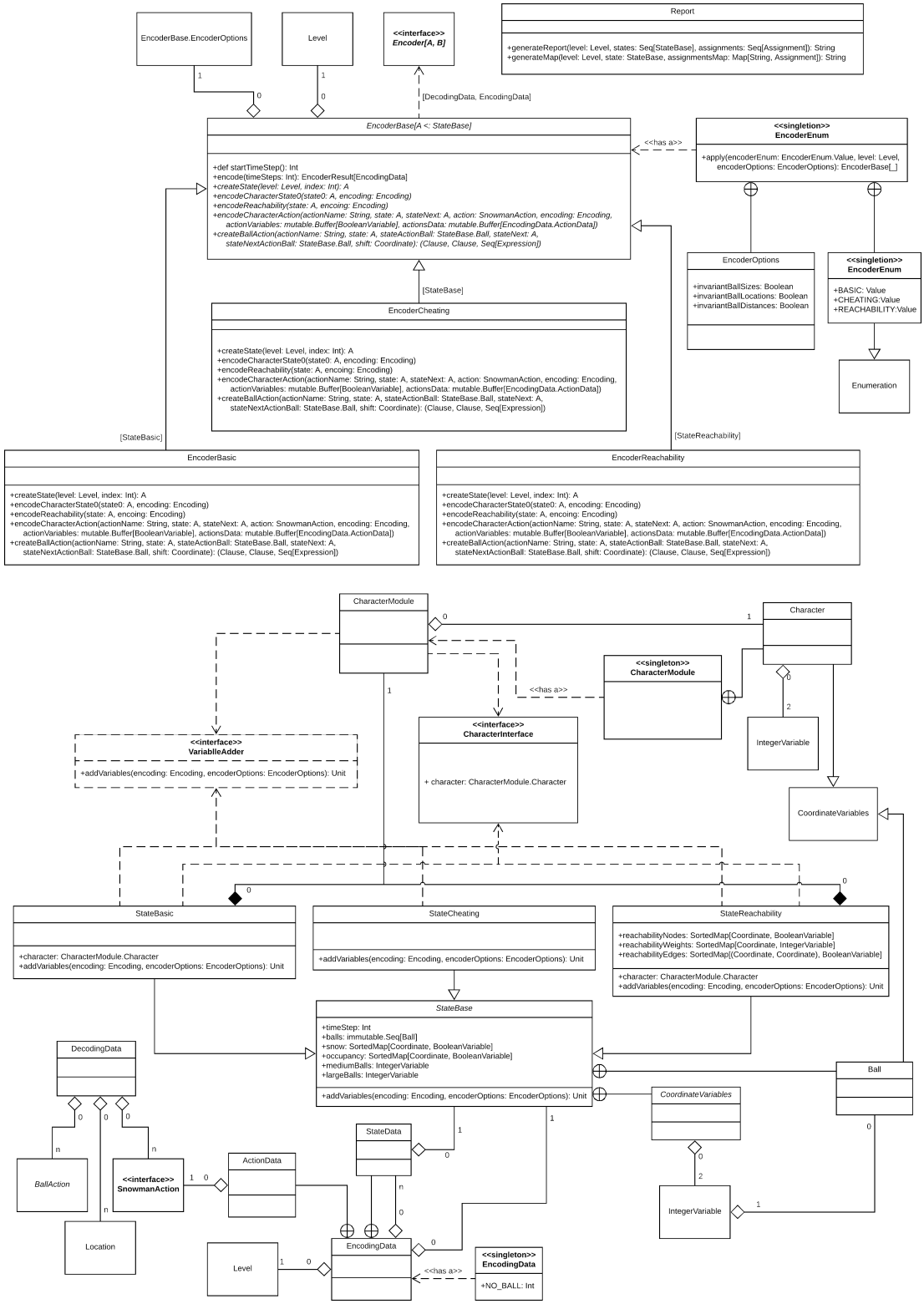


Figure 8.10: Class diagram from the planner planner package from Snowman problem module

Figure 8.11: Some screenshots examples from the graphical interface

This module is completely independent from the other modules which allows to change the graphical interface without any code modification.

8.2 PDDL Approach

The PDDL approach aims to solve the *Snowman problem* by using the PDDL standard Artificial Intelligence planning language. Because of the limitations with PDDL and its planers, two different approaches have been taken.

8.2.1 *adl* approach

The main idea behind the *adl* approach is to associate a distinct object to each possible locations of the level at hand, and to use predicates to define the adjacencies between them, i.e., the object *loc-1-1* represents the level coordinate $1, 1$. The predicates are:

(snow ?l - location) Is true if location *?l* has snow

(next ?from ?to - location ?dir - direction) Is true if locations *?from* and *?to* are adjacents and align in the direction *?dir*

(occupancy ?l - location) Is true if a location *?l* is occupied by a ball

(character-at ?l - location) Is true if the character is at location *?l*

(ball-at ?b - ball ?l - location) Is true if a ball *?b* is at location *?l*

(ball-size-small ?b - ball) Is true if the ball *?b* size is small

(ball-size-medium ?b - ball) Is true if the ball *?b* size is medium

(ball-size-large ?b - ball) Is true if the ball *?b* size is large

(goal) Is true if all the goal restrictions are meet

There are only two actions but the action *move_ball* uses *conditional-effects*:

move_character (?from ?to - location ?dir - direction Move the character from location *?from* to location *?to* and both locations are align in the direction *?dir*

move_ball (?b - ball ?ppos ?from ?to - location ?dir - direction) Move the ball from location *?from* to location *?to* if the player is at the location *?ppos* and all three locations are align in the directoin *?dir*. Update the snow, ball size and move the character according to the game rules.

The predicate occupancy is used to check if the character can be moved to its destination.

The *conditional-effects* could be removed by unfolding each conditional effect in a new action with the conditional statement as a new predicate and the effect as main effect.

8.2.1.1 The *move_ball* action

Here follows the *move_ball* action definition:

Precondition Description of the different statements used in the precondition:

Locations must be aligned.

```
(next ?ppos ?from ?dir)
(next ?from ?to ?dir)
```

The character must be behind the ball. Encodes the game rule number 1.

```
(ball-at ?b ?from)
(character-at ?ppos)
```

All the balls being at the same location as the ball which is being moved must be bigger. Encodes the rule number 4.

```
(forall (?o - ball)
    (or
        (= ?o ?b)
        (or
            (not (ball-at ?o ?from))
            (or
                (and
                    (ball-size-small ?b)
                    (ball-size-medium ?o))
                (and
                    (ball-size-small ?b)
                    (ball-size-large ?o))
                (and
                    (ball-size-medium ?b)
                    (ball-size-large ?o))))))
```

If there is a ball at the same location as the ball which is being moved, it can not be a ball in front. Encodes the rule number 3 and 4.

```
(or
    (forall (?o - ball)
        (or
            (= ?o ?b)
            (not (ball-at ?o ?from))))
    (forall (?o - ball)
            (not (ball-at ?o ?to))))
```

If there is a ball in front of the ball which is being moved, it has to be larger. Encodes the rule number 4.

```
(forall (?o - ball)
    (or
        (not (ball-at ?o ?to))
        (or
            (and
                (ball-size-small ?b)
                (ball-size-medium ?o))
            (and
                (ball-size-small ?b)
                (ball-size-large ?o))
            (and
                (ball-size-medium ?b)
                (ball-size-large ?o))))))
```

Notice that the structure *(or (not (A)) (B))* is a logical implication.

Effect Description of the different statements used in the effect:

The goal is true if all the balls are at the same location.

```
(when
    (forall (?o - ball)
        (or (= ?o ?b)
        (ball-at ?o ?to)))
    (goal))
```

Update the occupancy predicate according to the ball movement.

```
(not (occupancy ?from))
(occupancy ?to)
```

Update the ball location.

```
(not (ball-at ?b ?from))
(ball-at ?b ?to)
```

Move the character to the new location only if there was no ball under.

```
(when
    (forall (?o - ball)
        (or
            (= ?o ?b)
            (not (ball-at ?o ?from))))
    (and
        (not (character-at ?ppos))
        (character-at ?from)))
```

It removes the snow. It is always removed.

```
(not (snow ?to))
```

If there was snow, it increases the ball size.

```
(when
    (and
        (snow ?to)
        (ball-size-small ?b))
    (and
        (not (ball-size-small ?b))
        (ball-size-medium ?b)))
(when
    (and
        (snow ?to)
        (ball-size-medium ?b))
    (and
        (not (ball-size-medium ?b))
        (ball-size-large ?b))))
```

The full *adl* PDDL model can be found in Appendix 14.2.

An unfolded action can be seen as generating a new action for every combination parameters, referencing the concrete object when the parameter is being used. The main problem with this approach is the number of unfolded action which, because is to large, makes the search impossible. For example, the Andy instance, one of the smallest instances, has 3 balls and 25 playable locations. The number of unfolded actions would be:

move_character (?from ?to - location ?dir - direction) $25\ (?from) * 25\ (?to) * 4\ (?dir) = 2500$

move_ball (?b - ball) $3\ (?ball) * 25\ (?ppos) * 25\ (?from) * 25\ (?to) * 4\ (?dir) = 187500$

In total 190000 actions to explore every state, in contrast of the 4 actions the game has.

Another possible approach would substitute all the parameters by *conditional-effects*, but this approach is not possible since PDDL does not support nested *conditional-effects*.

8.2.2 object-fluents approach

The *object-fluents* approach aims to reduce the number of actions by removing all the location and directions from the action parameters. They can be removed because a new set of functions which saves the character and ball locations are introduced. Also, a new set of *next* functions are introduced which acts like the old *next* predicate. The new functions are:

(character-at) - location Returns the object location which is the character

(ball-at ?b - ball) - location Returns the object location which is the ball *?b*

(next-up ?l - location) - location Returns the object location which is in from of location *?l*

(next-down ?l - location) - location Returns the object location which is behind of location *?l*

(next-right ?l - location) - location Returns the object location which is in to right of location *?l*

(next-left ?l - location) - location Returns the object location which is in to left of location *?l*

Because the direction is removed from the action parameter, now there will be 4 actions of each kind, one for each direction. The number of unfolded actions for the Andy instance becomes just 16:

move_direction_ character $4\ actions(4\ directions)$

move_direction_ball (?b - ball) $4\ actions * 3\ (?ball) = 12$

The full *object-fluents* PDDL model can be found in Appendix 14.3.

The main problem of this description is that the only planner that supports this subset is *Rantanplan*[10]. This planner uses planning as SMT approach and supports this PDDL specification. However the initial experiments with a basic configuration have been quite disappointing since it was not able to solve the simplest game instance. Further configurations of the solver should be considered to try to solve real instances. This is left as further work.

8.2.3 Using a planner to solve the problem: The reality

Once the domain and the problem descriptions are generated, they can be used as input to a planner. Multiple planners from the last *IPC* (2018) have been tested but none of them could solve a single instance. Most of the planners were based on the planner *Fast Downward*[13], which seems to have a problem interpreting the *Snowman problem* domain, which makes the planner to not halt. Other planners simply did return an invalid plan.

Because searching for newer planners is a very time consuming task, this part (and possible results) has been left as further work.

8.3 SMT Approach

The SMT approach allows to find an optimal solution to the *Snowman problem*. This is desired since the main objective of the project is to find undesired solutions, which usually are easier solutions that the wanted solution.

During the encoding development process three different encodings have been created since each one has its advantages an inconveniences. The theory used is Linear Integer Arithmetic since integer variables and linear arithmetic expressions are required.

8.3.1 Basic encoding

A first approach to encode the *Snowman problem* is to encode all possible actions a player can do in the game. Even though there are only four actions in the game (up, down, left, right)[1], since they have different effects, these actions can be split into two categories: character movement actions and ball movement actions.

This way, this first encoding encodes four actions for the character movement actions and four actions for the ball movement actions, but for each ball. To perform a ball movement action, first the player has to travel next to the ball using the character movements actions. Example 8.3.3 shows the solution found for the Tanya problem with the *Basic encoding*.

The preliminary result hovever, showed that this first encoding was not able to solve many levels in a reasonable time.

8.3.1.1 Instance

Each problem instance has the following information:

- Set of locations of the scenario: Pos.
- Initial character location: $Character$.

[1]The game is played using the keyboard arrows

- Set of balls with their initial locations and sizes: $Balls$.
- Set of initial floor locations with snow: $Snow$.
- Set of wall locations: $Walls$.

A location is a coordinate, i.e., a pair of natural numbers (x, y).

For the sake of readability of the following encoding, the set $BallPos$ is defined as $(i, j) \in Pos$ such that there exists $b \in Balls$ with $b.x = i \wedge b.y = j$.

8.3.1.2 State

The game state at time step t will be represented by the following state variables $\langle c, b, s, o, ibs \rangle$:

- Character coordinates variables. Integer variables for each time step t: $c^t.x$ and $c^t.y$.
- Set of ball variables. Integer variables for each time step t and ball k: $b_k^t.x, b_k^t.y$ and $b_k^t.size$
- Set of snow variables that represent if each coordinate has snow. Boolean variables for each time step t, and coordinates x, y: $s_{x,y}^t$
- Set of occupancy variables that represent if there's a ball or a wall. Boolean variables for each time step t, and coordinates x, y: $o_{x,y}^t$
- For the invariant *INVARIANT_BALL_SIZES*, and if the invariant is enabled, for each time step t, a medium size balls counter ibs_m^t and a large size balls counter ibs_l^t

A state S^t will be a valuation over the sets of state variables of time step t.

8.3.1.3 Initial state

In the encoding the initial state is defined, according to the instance data, by the following formula, see Example 8.3.1

$$c^0.x = Character.x \quad \wedge \quad c^0.y = Character.y$$

$$\bigwedge_{k \in Balls} (b_k^0.x = Balls_k.x \quad \wedge \quad b_k^0.y = Balls_k.y \quad \wedge \quad b_k^0.size = Balls_k.size)$$

$$\bigwedge_{(i,j) \in Snow} s_{i,j}^0$$

$$\bigwedge_{(i,j) \in (Pos \setminus Snow)} \neg s_{i,j}^0$$

$$\bigwedge_{(i,j) \in (Walls \cup BallPos)} o_{i,j}^0$$

$$\bigwedge_{(i,j) \in (Pos \setminus (Walls \cup BallPos))} \neg o_{i,j}^0$$

Example 8.3.1. *State S^0 form the Tanya level:*

$c^0.x = 1$
$c^0.y = 3$

$b_0^0.x = 2$
$b_0^0.y = 2$
$b_0^0.s = 1$

$b_1^0.x = 2$
$b_1^0.y = 3$
$b_1^0.s = 1$

$b_2^0.x = 2$
$b_2^0.y = 4$
$b_2^0.s = 1$

$o_{1,0}^0.x = true$
$o_{2,0}^0.x = true$
$o_{3,0}^0.x = true$
$o_{0,1}^0.x = true$
$o_{1,1}^0.x = false$
$o_{2,1}^0.x = false$
$o_{3,1}^0.x = false$
$o_{4,1}^0.x = true$
$o_{0,2}^0.x = true$
$o_{1,2}^0.x = false$
$o_{2,2}^0.x = true$
$o_{3,2}^0.x = false$
$o_{4,2}^0.x = true$
$o_{0,3}^0.x = true$
$o_{1,3}^0.x = false$
$o_{2,3}^0.x = true$

$o_{3,3}^0.x = false$
$o_{4,3}^0.x = true$
$o_{0,4}^0.x = true$
$o_{1,4}^0.x = false$
$o_{2,4}^0.x = true$
$o_{3,4}^0.x = false$
$o_{4,4}^0.x = true$
$o_{0,5}^0.x = true$
$o_{1,5}^0.x = false$
$o_{2,5}^0.x = false$
$o_{3,5}^0.x = false$
$o_{4,5}^0.x = true$
$o_{1,6}^0.x = true$
$o_{2,6}^0.x = true$
$o_{3,6}^0.x = true$

$s_{1,1}^0.x = true$
$s_{2,1}^0.x = true$
$s_{3,1}^0.x = true$
$s_{1,2}^0.x = true$
$s_{3,2}^0.x = true$
$s_{1,3}^0.x = true$
$s_{3,3}^0.x = true$
$s_{1,4}^0.x = true$
$s_{3,4}^0.x = true$
$s_{1,5}^0.x = true$
$s_{2,5}^0.x = true$
$s_{3,5}^0.x = true$

8.3.1.4 Actions

The encoding has four character movement actions

- $move\_character\_up$
- $move\_character\_down$
- $move\_character\_left$
- $move\_character\_right$

and, for each ball k, four ball actions

- $move\_ball\_up_k$
- $move\_ball\_down_k$
- $move\_ball\_left_k$
- $move\_ball\_right_k$

For each action w and each time step t, it is encoded that, if the action is executed, the preconditions that must be satisfied by state S^t and the effects that must be met at state S^{t+1}. Therefore, there will be a Boolean variable meaning whether that action is executed or not at that time step:

$$a^t_{move\_up\_character}, a^t_{move\_down\_character}, ..., a^t_{move\_up\_ball_k}, a^t_{move\_down\_ball_k}, \cdots$$

Moreover, the following formulas will be added to the encoding, dealing with preconditions and effects of action w:

$$EFF^{t+1} \leftrightarrow a_w^t$$
$$a_w^t \rightarrow PRE^t$$

For the sake of space, only the EFF and PRE formulas of the *move character* and the *move ball* action upwards will be formulated.

8.3.1.5 Character movement actions

Action *move_up_character*, i.e., move the character one location upwards.

Action preconditions:

$$PRE^t = CHARACTER\_LOCATION\_VALID^t$$

Action effects:

$$\begin{aligned} EFF^{t+1} = {} & MOVE\_CHARACTER^{t+1} \quad \wedge \\ & EQUAL\_BALLS\_VARIABLES^{t+1} \quad \wedge \\ & EQUAL\_SNOW\_VARIABLES^{t+1} \quad \wedge \\ & EQUAL\_OCCUPANCY\_VARIABLES^{t+1} \end{aligned}$$

$CHARACTER\_LOCATION\_VALID^t$ The next character location has not to be occupied.

$$\bigvee_{(i,j)\in Pos} (\ c^t.x = i \quad \wedge \quad c^t.y = j \quad \wedge \quad \neg o_{i,j+1}^t \ )$$

$MOVE\_CHARACTER^{t+1}$ The character's location in next time step is the same as the current one but one, location up.

$$c^t.x = c^{t+1}.x \quad \wedge \quad c^t.y + 1 = c^{t+1}.y$$

$EQUAL\_BALLS\_VARIABLES^{t+1}$ All balls have the same coordinates and size in the current and the next time step.

$$\bigwedge_{k\in Balls} (b_k^t.x = b_k^{t+1}.x \quad \wedge \quad b_k^t.y = b_k^{t+1}.y \quad \wedge \quad b_k^t.size = b_k^{t+1}.size)$$

$EQUAL\_SNOW\_VARIABLES^{t+1}$ Maintains all snow variables between time step t and $t+1$

$$\bigwedge_{(i,j)\in Snow} s_{i,j}^t = b_k i, j^{t+1}$$

$EQUAL\_OCCUPANCY\_VARIABLES^{t+1}$ Maintains all snow variables between time step t and $t+1$

$$\bigwedge_{(i,j)\in Pos\setminus Walls} o_{i,j}^t = oi, j^{t+1}$$

8.3.1.6 Ball movement actions

Action $move\_up\_ball_A$, i.e., moving a single ball A one location upwards.

Action preconditions:

$$\begin{aligned} PRE^t = {} & CHARACTER\_NEXT\_TO\_BALL^t \quad \wedge \\ & NO\_WALL\_IN\_FRONT^t \quad \wedge \\ & NO\_OTHER\_BALLS\_OVER^t \quad \wedge \\ & \neg(OTHER\_BALL\_IN\_FRONT^t \quad \wedge \quad OTHER\_BALL\_UNDER^t) \quad \wedge \\ & OTHER\_BALLS\_IN\_FRONT\_LARGER^t \end{aligned}$$

Action effects:

$$\begin{aligned} EFF^{t+1} = {} & MOVE\_BALL^{t+1} \quad \wedge \\ & (\neg OTHER\_BALL\_UNDER^t \quad \rightarrow \quad MOVE\_CHARACTER^{t+1}) \quad \wedge \\ & (OTHER\_BALL\_UNDER^t \quad \rightarrow \quad EQUAL\_CHARACTER\_VARIABLES^{t+1}) \quad \wedge \\ & EQUAL\_OTHER\_BALLS\_VARIABLES^{t+1} \quad \wedge \\ & UPDATE\_SNOW\_VARIABLES^{t+1} \quad \wedge \\ & UPDATE\_BALL\_SIZE^{t+1} \quad \wedge \\ & UPDATE\_OCCUPANCY\_VARIABLES^{t+1} \quad \wedge \\ & INVARIANT\_BALL\_SIZES^{t+1} \end{aligned}$$

$CHARACTER\_NEXT\_TO\_BALL^t$ The character has to be next to the ball that it will move. Namely, it has to be at the same x coordinate and at one location less regarding to coordinate y.

$$c^t.x = b^t_A.x \quad \wedge \quad c^t.y = b^t_A.y - 1$$

$NO\_WALL\_IN\_FRONT^t$ The ball A location one location less regarding to coordinate y is playable.

$$\bigvee_{(i,j+1) \in Pos \setminus Walls} (b^t_A.x = i \quad \wedge \quad b^t_A.y = j)$$

$NO\_OTHER\_BALLS\_OVER^t$ The ball A is the smallest in that location, hence it is the ball on top for that location.

$$\bigwedge_{k \in Balls, k \neq A} (b^t_A.x \neq b^t_k.x \quad \vee \quad b^t_A.y \neq b^t_k.y \quad \vee \quad b^t_A.size < b^t_k.size)$$

$OTHER\_BALLS\_IN\_FRONT^t$ There is a ball in front of ball A.

$$\bigvee_{k \in Balls, k \neq A} (b^t_A.x = b^t_k.x \quad \wedge \quad b^t_A.y + 1 = b^t_k.y)$$

$OTHER\_BALL\_UNDER^t$ There is a ball of bigger size than A in A's location, i.e., there is a ball under A.

$$\bigvee_{k \in Balls, k \neq A} (b^t_A.x = b^t_k.x \quad \wedge \quad b^t_A.y = b^t_k.y \quad \wedge \quad b^t_A.size < b^t_k.size)$$

$OTHER\_BALLS\_IN\_FRONT\_LARGER^t$ Ball A is smaller than any ball in front of it.

$$\bigwedge_{k \in Balls, k \neq A} (b_A^t.x \neq b_k^t.x \quad \vee \quad b_A^t.y + 1 \neq b_k^t.y \quad \vee \quad b_A^t.size < b_k^t.size)$$

$MOVE\_BALL^{t+1}$ Ball's A location in the next time step is the same as the current one but one location up.

$$b_A^t.x = b_A^{t+1}.x \quad \wedge \quad b_A^t.y + 1 = b_A^{t+1}.y$$

$EQUAL\_CHARACTER\_VARIABLES^{t+1}$ Character's location in the next time step is the same as the current one.

$$c^t.x = c^{t+1}.x \quad \wedge \quad c^t.y = c^{t+1}.y$$

$EQUAL\_OTHER\_BALLS\_VARIABLES^{t+1}$ All balls except A have the same coordinates and size in the current and the next time step.

$$\bigwedge_{k \in Balls, k \neq A} (b_k^t.x = b_k^{t+1}.x \quad \wedge \quad b_k^t.y = b_k^{t+1}.y \quad \wedge \quad b_k^t.size = b_k^{t+1}.size)$$

$UPDATE\_SNOW\_VARIABLES^{t+1}$ A location has snow in the next time step if and only if it has snow and ball A will not be at that location on the next time step.

$$\bigwedge_{(i,j+1) \in Snow} \Big((\ s_{i,j+1}^t \quad \wedge \quad (b_A^t.x \neq i \quad \vee \quad b_A^t.y \neq j)\ ) \quad \leftrightarrow \quad s_{i,j+1}^{t+1} \Big)$$

$UPDATE\_BALL\_SIZE^{t+1}$ If there is snow on the location up

$$(\ \bigvee_{(i,j+1) \in Snow} b_A^t.x = i \quad \wedge \quad b_A^t.y = j \quad \wedge \quad s_{i,j+1}^t) \quad \leftrightarrow \quad s$$

increase ball's A size

$$\big((s \quad \wedge \quad b_A^t.s = 1) \quad \rightarrow \quad b_A^{t+1}.s = 2\big) \quad \wedge \quad \big((s \quad \wedge \quad b_A^t.s = 2) \quad \rightarrow \quad b_A^{t+1}.s = 4\big) \quad \wedge \quad \big(\neg s \quad \vee \quad (b_A^t.s = 4) \quad \rightarrow \quad b_A^{t+1}.s = b_A^t.s\big)$$

$UPDATE\_OCCUPANCY\_VARIABLES^{t+1}$ A location is occupied if and only if there is a ball in it. For a graphical example, see Example 8.3.2.

$$\bigwedge_{(i,j) \in Pos \setminus Walls} (\ \bigvee_{k \in Balls} (\ b_k^t.x = i \quad \wedge \quad b_k^{t+1}.y = j)\ ) \quad \leftrightarrow \quad o_{i,j}^t$$

$INVARIANT\_BALL\_SIZES^{t+1}$ If the ball A has changed its size from or to medium size, update the medium size state counter. Only is added if the invariant is enabled.

$$(b_A^t.size \neq 2 \quad \wedge \quad b_A^t.size = 2) \quad \rightarrow (ibs_m^t + 1 = ibs_m^{t+1}) \quad \wedge$$
$$(b_A^t.size = 2 \quad \wedge \quad b_A^t.size \neq 2) \quad \rightarrow (ibs_m^t - 1 = ibs_m^{t+1})$$

If the ball A has changed its size from or to large size, update the large size state counter.

$$\begin{aligned}(b_A^t.size \neq 4 \quad \wedge \quad b_A^t.size = 4) \quad &\rightarrow (ibs_l^t + 1 = ibs_l^{t+1}) \quad \wedge \\ (b_A^t.size = 4 \quad \wedge \quad b_A^t.size \neq 4) \quad &\rightarrow (ibs_l^t - 1 = ibs_l^{t+1})\end{aligned}$$

Since it is impossible to make a ball smaller, for each snowman, there has to be at least one small ball and there can not be two big balls. For all time steps s, the medium ball counter must be smaller than $2 * |Balls| + 1$ be smaller than the large ball counter must be smaller than $2 * |Balls|$.

$$\begin{aligned}ibs_m^{t+1} < 2 * |Balls| + 1 \quad \wedge \\ ibs_l^{t+1} < 2 * |Balls|\end{aligned}$$

8.3.1.7 Occupancy variables update

For all time steps t, occupancy variables for wall locations will always be true. For a graphical example, see Example 8.3.2.

$$\bigwedge_{(i,j) \in Walls} o_{i,j}^t$$

8.3.1.8 Goal

The goal is defined by the following formula over the state variables of S^n. In case of having an instance with only one snowman, the following clause will be added:

$$b_1^n.x = b_2^n.x \quad \wedge \quad b_1^n.y = b_2^n.y \quad \wedge \quad b_1^n.x = b_3^n.x \quad \wedge \quad b_1^n.y = b_3^n.y$$

If the instance has more than one snowman, a new location variable x_i, y_i will be created for each snowman i. Each ball has to be in one of this new locations. The preconditions in the ball movement actions will force that only three balls are at the same location.

$$\bigwedge_{k \in Balls} (\bigvee_i b_k^n.x = x_i \quad \wedge \quad b_k^n.y = y_i)$$

8.3.1.9 Invariants

An invariant is a clause which does not add meaningful effects but reduces the search space. Encoding an invariant sometimes can give worse results since there must be a balance between the invariant solving cost and the reduction in the search space.

Currently, the invariant *INVARIANT_BALL_SIZES* is optional since it does not improve the performance.

Example 8.3.2. *Graphical visualization of variables assignments from the Andy level using the* Basic encoding.

```
STATE 0
Map      Snow     Occupancy
-#####-  -------  -11111-
#.....#  -11111-  1000001
#.1.1'#  -1-1---  1010101
#..q''#  -111---  1000001
#.1'''#  -1-----  1010001
#'''..#  ----11-  1000001
-#####-  -------  -11111-
```

```
STATE 1
Map      Snow     Occupancy
-#####-  -------  -11111-
#.....#  -11111-  1000001
#.1.1'#  -1-1---  1010101
#...p'#  -111---  1000001
#.1'''#  -1-----  1010001
#'''..#  ----11-  1000001
-#####-  -------  -11111-
```

```
STATE 2
Map      Snow     Occupancy
-#####-  -------  -11111-
#.....#  -11111-  1000001
#.1.1'#  -1-1---  1010101
#...'p#  -111---  1000001
#.1'''#  -1-----  1010001
#'''..#  ----11-  1000001
-#####-  -------  -11111-
```

...

```
STATE 17
Map      Snow     Occupancy
-#####-  -------  -11111-
#.....#  -11111-  1000001
#.''''#  -1-0---  1000001
#.'6''#  -100---  1001001
#.'1''#  -1-----  1001001
#'p'..#  ----11-  1000001
-#####-  -------  -11111-
```

```
STATE 18
Map      Snow     Occupancy
-#####-  -------  -11111-
#.....#  -11111-  1000001
#.''''#  -1-0---  1000001
#.'6''#  -100---  1001001
#.'1''#  -1-----  1001001
#''p..#  ----11-  1000001
-#####-  -------  -11111-
```

```
STATE 19
Map      Snow     Occupancy
-#####-  -------  -11111-
#.....#  -11111-  1000001
#.''''#  -1-0---  1000001
#.'7''#  -100---  1001001
#.'p''#  -1-----  1000001
#'''..#  ----11-  1000001
-#####-  -------  -11111-
```

Map symbols description:

- **-** *no variable defined*
- **#** *Wall*
- **p** *Character*
- **q** *Character with on the floor*
- **1** *Small ball*
- **2** *Medium ball*
- **4** *Large ball*
- **'** *grass*
- **.** *snow*

Any number not defined is the sum of the ball symbols.

Occupancy and snow symbols description:

- **-** *no variable defined*
- **0** *false*
- **1** *true*

Example 8.3.3. *Solver solution from Tanya level using the* Basic encoding. *Returns 17 actions of which 5 are ball movement actions.*

Output	***Action***
solver	*up*
	right
	up
	right
	down
	left
	left
	down
	down
	right
	down
	right
	up
	left
	left
	up
	right

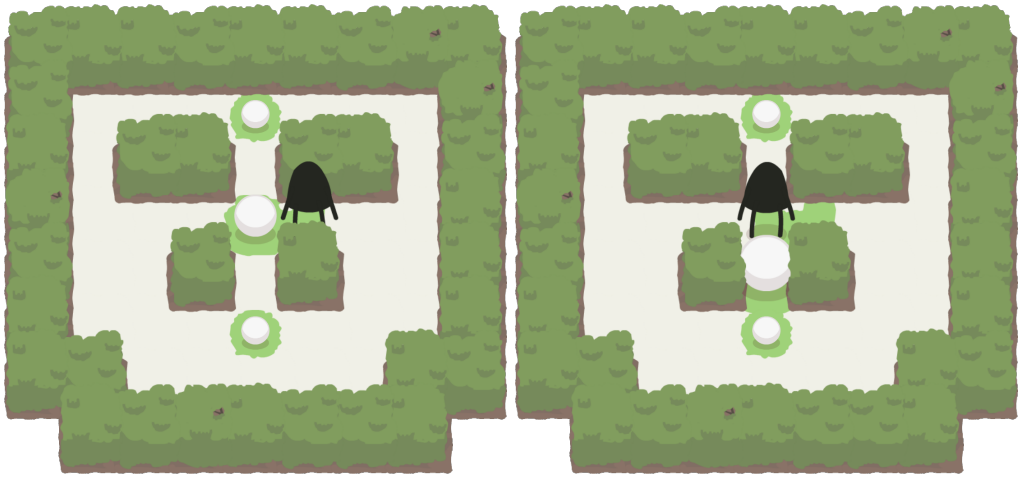
8.3.2 Cheating encoding

A big number of actions that are realized in the game are character movement actions, since when the character move a ball, usually it blocks a path and the character is forced to get around the level, see for instance Example 8.3.4.

Example 8.3.4. *The Willow level, which can be solved with 83 actions, only needs 14 ball movement actions.*

This encoding is based on the previous encoding, but it removes the character movement actions allowing the character to teleport anywhere on the map as long as is not occupied. This can be seen as cheating, as a ball can block a path to a candidate ball movement action, see for instance Example 8.3.5 where it is shown that the character moves the ball of the middle without despite the ball above is blocking its path.

Example 8.3.5. *Willow cheating actions obtained from the* Cheating encoding.



Example 8.3.8 shows the solution found for the Tanya problem with the *Cheating encoding*.

8.3.2.1 State

Now the state variables are just $\langle b, s, o, ibs \rangle$. It does not contain the character variables since they can be deduced by knowing which ball action movement has been executed.

8.3.2.2 Ball movement actions

Action $move\_up\_ball_A$, i.e., moving a single ball A one location upwards. By allowing the character to teleport, it only has to check if the teleported location is valid. The previous model precondition gets slightly modified.

New action precondition:

$$\begin{aligned} PRE^t = {} & NO\_WALL\_IN\_FRONT^t \quad \wedge \\ & NO\_OTHER\_BALLS\_OVER^t \quad \wedge \\ & \neg(OTHER\_BALL\_IN\_FRONT^t \quad \wedge \quad OTHER\_BALL\_UNDER^t) \quad \wedge \\ & OTHER\_BALLS\_IN\_FRONT\_LARGER^t \quad \wedge \\ & CHARACTER\_LOCATION\_TELEPORT\_VALID^t \end{aligned}$$

New action effects:

$$\begin{aligned} EFF^{t+1} = {} & MOVE\_BALL^{t+1} \quad \wedge \\ & EQUAL\_OTHER\_BALLS\_VARIABLES^{t+1} \quad \wedge \\ & EQUAL\_SNOW\_VARIABLES^{t+1} \quad \wedge \\ & UPDATE\_BALL\_SIZE^{t+1} \quad \wedge \\ & UPDATE\_OCCUPANCY\_VARIABLES^{t+1} \quad \wedge \\ & INVARIANT\_BALL\_SIZES^{t+1} \end{aligned}$$

$CHARACTER\_LOCATION\_TELEPORT\_VALID^t$ Character location has to be not occupied. It can be seen as ball A south location has to be not occupied.

$$\bigvee_{(i,j-1)\in Pos} (b_A^t.x = i \quad \wedge \quad b_A^t.y = j \quad \wedge \quad \neg o_{i,j-1}^t)$$

8.3.2.3 Path reconstruction

Because the solver only returns an abstract plan consisting of the ball movement actions, the plan is incomplete and must be reconstructed to be able to fully reproduce it, in particular we need all movements of the character between ball movements. Here, a post-processing is realized which reconstructs the required paths and the corresponding actions to do them using the $A*$ algorithm, see Example 8.3.11.

Example 8.3.6. *Debug information from the path-reconstruction process. The origin from the reference system is the bottom left:*

```
STATE
-#######-
#...1...#
#.##.##.#
#...'2..#
#..#.#..#
#...1...#
##.....##
-#######-

STATE NEXT
-#######-
#...1...#
#.##.##.#
#...2'..#
#..#.#..#
#...1...#
##.....##
-#######-
```

```
Path Start: Coordinate(4,4)
Path End: Coordinate(6,4)
    Path: left
    Path: left
    Path: down
    Path: down
    Path: right
    Path: down
    Path: right
    Path: right
    Path: up
    Path: right
    Path: up
    Path: up
Action: left
```

Map symbols description:

- **-** *no variable defined*
- **#** *Wall*
- **p** *Character*
- **q** *Character with on the floor*
- **1** *Small ball*
- **2** *Medium ball*
- **4** *Large ball*
- **3** *Small ball on top of a medium ball*
- **5** *Small ball on top of a large ball*
- **6** *Medium ball on top of a large ball*
- **7** *Small ball on top of a medium ball on top of a large ball*
- **'** *grass*
- **.** *snow*

But, since the encoding is potentially cheating, it is not always possible to reconstruct the path. If a path can not be reconstructed, it still will return path, which will be invalid because it will allow to pass through the balls. Example 8.3.11 illustrates an invalid path reconstruction in a particular configuration of the Willow problem.

Example 8.3.7. *Debug information from the previous Example 8.3.5, which failed the path-reconstruction process.*

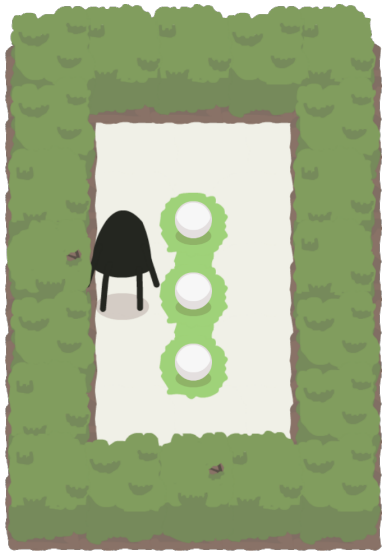
```
STATE
-#######-
#...1...#
#.##.##.#
#...2'..#
#..#.#..#
#...1...#
##.....##
-#######-

STATE NEXT
-#######-
#...1...#
#.##.##.#
#...''..#
#..#4#..#
#...1...#
##.....##
-#######-
```

```
Path Start: Coordinate(5,4)
Path End: Coordinate(4,5)
Omitting occupancy for balls
    Path: left
    Path: up
Action: down
```

Example 8.3.8. *Solver solution from Tanya level using the* Cheating encoding. *Returns 21 actions,at which 5 are ball movement actions:*

Output	***Action***
pathfinder	*up*
	up
	right
	right
	down
solver	*left*
pathfinder	*up*
	left
solver	*down*
pathfinder	*right*
	right
	down
	down
solver	*left*
pathfinder	*down*
	left
solver	*up*
pathfinder	*right*
	right
	up
solver	*left*



8.3.3 Reachability encoding

This encoding is based on the previous *Cheating encoding*, but it introduces a new reachability precondition. These preconditions ensure that there is a path from the position of the character when doing a ball movement action to the position of the character to perform the next ball movement action. The enconding also reintroduces the character position variables.

Each location represents a graph's node which is connected to the adjacent locations. Each pair of adjacent nodes will have two edges, an incoming and outgoing edge, to conform an undirected graph. A node will be reachable if there is the character on that location or and adjacent node is reachable.

Example 8.3.12 shows the solution found for the Tanya problem with the *Reachability encoding*.

8.3.3.1 Reachability precondition

The reachability precondition encodes an *s-t-reachability* problem extending the *Topological Sort with Indices* described in [14]. Is important to remark that in contrast to [14], this extension must deal with non-static graphs in the sense that it is not know, when the the encoding is being build, which will be the source and which will be the target of the s-t-reachability problem. Neither is not known which positions will be occupied. Therefore, we believe that this is a remarkable contribution of this project.

The encoding introduces the following variables:

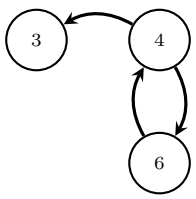
- A Boolean variable rn_i per node i stating node reachability
- A Boolean variable re_{o-d} per (directed) edge $o \to d$ stating whether this edge is used in the path souce-target
- An integer variable *source* whose value corresponds to the node that is the starting node

and clauses:

- target node is reachable: unary rn_t
- for all nodes i, if that node is not the source and is reachable, then one of the edges pointing to it must be used in the path: $(source \neq i \wedge rn_i) \to (re_{n_1-i} \vee \cdots \vee re_{n_k-i})$
- for all edges $o \to d$, if that edge is used, its origin must be reachable: $re_{o-d} \to rn_o$

With this encoding, the edge variables set to true are showing a possible path from source to target. However this encoding does not work in the presence of cycles in the graph as example 8.3.9 shows.

Example 8.3.9. *Given the following graph with cycles with the source node being* 3 *and the destination node being* 6*:*



the encoding follows: The destination node 6 *is reachable:*

$$rn_6$$

the node 3 *is the source:*

$$source = 3$$

if a node is reachable and is not the source, then some incoming edge must provide it its reachability:

$$(source \neq 3 \wedge rn_3) \rightarrow re_{4-3}$$
$$(source \neq 4 \wedge rn_4) \rightarrow re_{6-4}$$
$$(source \neq 6 \wedge rn_6) \rightarrow re_{4-6}$$

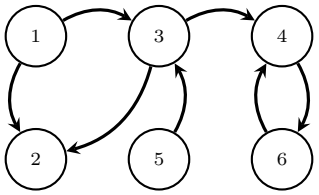
if an edge has been used to provide reachability, its origin must be reachable:

$$re_{4-6} \rightarrow rn_4$$
$$re_{6-4} \rightarrow rn_6$$
$$re_{4-3} \rightarrow rn_4$$

For example, if the SAT solver states the variables rn_4, rn_6, re_{4-6} and re_{6-4} to true, the formula is satisfied though clearly node 6 is not reachable from node 3.

For a graph with cycles, an additional clause is needed to avoid the reachable clauses to be supported by a cyclic path non coming from the source. To forbid cycles in the paths, we introduce weight variables rw_i for each node i and we will force that the weight associated to the source of each selected edge has to be bigger than the one of the destination, see Example 8.3.10.

Example 8.3.10. *Given the following graph:*



and assuming that the unknown destination node is i, the reachability encoding is as follows:

The destination node i is reachable:

$$rn_i$$

some node j is the source:

$$source = j$$

if a node is reachable and is not the source, then some incoming edge must provide it its reachability:

$$source = 1 \vee \neg rn_1$$
$$(source \neq 3 \wedge rn_3) \rightarrow (re_{1-3} \vee re_{5-3})$$
$$(source \neq 4 \wedge rn_4) \rightarrow (re_{3-4} \vee re_{6-4})$$
$$source = 5 \vee \neg rn_5$$
$$(source \neq 6 \wedge rn_6) \rightarrow re_{4-6}$$

if an edge has been used to provide reachability, its origin must be reachable:

$$re_{1-3} \rightarrow rn_1$$
$$re_{1-2} \rightarrow rn_1$$
$$re_{3-2} \rightarrow rn_3$$
$$re_{3-4} \rightarrow rn_3$$
$$re_{4-6} \rightarrow rn_4$$
$$re_{5-3} \rightarrow rn_5$$
$$re_{6-4} \rightarrow rn_6$$

edges used to provide reachability must not form any cycle:

$$re_{1-2} \rightarrow rw_1 < rw_2$$
$$re_{1-3} \rightarrow rw_1 < rw_3$$
$$re_{3-2} \rightarrow rw_3 < rw_2$$
$$re_{3-4} \rightarrow rw_3 < rw_4$$
$$re_{4-6} \rightarrow rw_4 < rw_6$$
$$re_{5-3} \rightarrow rw_5 < rw_3$$
$$re_{6-4} \rightarrow rw_6 < rw_4$$

It is not difficult to see that if the source is node 2 *and the target is node* 6*, the cyclic path to provide reachability used in example 8.3.9 would make clauses* $re_{4-6} \rightarrow rw_4 < rw_6$ *and* $re_{6-4} \rightarrow rw_6 < rw_4$ *unsatisfiable.*

8.3.3.2 State

This encoding state variables can be represented by $\langle c, b, s, o, ibs, rn, re, rw \rangle$, where:

- Set of reachability nodes which represent if the location is reachable. Boolean variables for each time step t and coordinates x, y: $rn^t_{x,y}$
- Set of reachability edges used to propagate the reachability between nodes. Boolean variables for each time step t, source coordinates x, y and destination coordinates i, j: $re^t_{x,y,i,j}$
- Set of reachability nodes weights used to break the cyclicity. Integer variables for each time step t and coordinates x, y: $rw^t_{x,y}$

Each location (x, y) will have its reachable variable $rn^t_{x,y}$ and weight variable $rw^t_{x,y}$ and will have an edge variable for each adjacent location $re^t_{x,y,x+1,y}$, $re^t_{x,y,x-1,y}$, $re^t_{x,y,x,y+1}$ and $re^t_{x,y,x,y-1}$. Since it is an $s - t - reachability$ encoding, this encoding also needs the character location as starting location for $s - t - reachability$.

8.3.3.3 Ball movement actions

Action $move\_up\_ball_A$, i.e., moving a single ball A one location upwards.

New action precondition:

$$\begin{aligned} PRE^t = {} & NO\_WALL\_IN\_FRONT^t \quad \wedge \\ & NO\_OTHER\_BALL\_OVER^t \quad \wedge \\ & \neg(OTHER\_BALL\_IN\_FRONT^t \quad \wedge \quad OTHER\_BALL\_UNDER^t) \quad \wedge \\ & OTHER\_BALLS\_IN\_FRONT\_LARGER^t \quad \wedge \\ & CHARACTER\_LOCATION\_TELEPORT\_VALID^t \quad \wedge \\ & REACHABILITY^t \end{aligned}$$

New action effects:

$$\begin{aligned} EFF = {} & MOVE\_BALL^{t+1} \quad \wedge \\ & (\neg OTHER\_BALL\_UNDER^t \quad \rightarrow \quad TELEPORT\_CHARACTER\_BALL^{t+1}) \quad \wedge \\ & (OTHER\_BALL\_UNDER^t \quad \rightarrow \quad TELEPORT\_CHARACTER^{t+1}) \quad \wedge \\ & EQUAL\_OTHER\_BALLS\_VARIABLES^{t+1} \quad \wedge \\ & EQUAL\_SNOW\_VARIABLES^{t+1} \quad \wedge \\ & UPDATE\_BALL\_SIZE^{t+1} \quad \wedge \\ & UPDATE\_OCCUPANCY\_VARIABLES^{t+1} \quad \wedge \\ & INVARIANT\_BALL\_SIZES^{t+1} \end{aligned}$$

$REACHABILITY^t$ The location where it has to be the character to perform the movement ball action A is reachable.

$$\bigvee_{(i,j-1)\in Pos} (b_A^t.x = i \quad \wedge \quad b_A^t.y = j \quad \wedge \quad rn_{i,j-1}^t)$$

$TELEPORT\_CHARACTER\_BALL^{t+1}$ The character's location at time step $t+1$ is the same as ball A's location at time step t.

$$c^{t+1}.x = b_A^t.x \quad \wedge \quad c^{t+1}.y = b_A^t.y$$

$TELEPORT\_CHARACTER^{t+1}$ The character's location at time step $t+1$ is one location less regarding to coordinate y from the ball A's location at time step t.

$$c^{t+1}.x = b_A^t.x \quad \wedge \quad c^{t+1}.y = b_A^t.y - 1$$

8.3.3.4 Rechability clauses

If not the character location, a node is reachable thanks to some incoming edge from another node:

$$\bigwedge_{(i,j)\in Pos\setminus Walls} \Big(((c^t.x \neq i \quad \vee \quad c^t.y \neq j) \quad \wedge \quad rn_{i,j}^t) \quad \rightarrow \quad \big(re_{i+1,j,i,j}^t \quad \vee \quad re_{i-1,j,i,j}^t \quad \vee \quad re_{i,j+1,i,j}^t \quad \vee \quad re_{i,j-1,i,j}^t\big)\Big)$$

the edge "used to provide reachability" must come from a reachable node.

$$\bigwedge_{(i,j)\in Pos\setminus Walls} \big((re^t_{i,j,i+1,j} \rightarrow rn^t_{i,j}) \quad \wedge \quad (re^t_{i,j,i-1,j} \rightarrow rn^t_{i,j}) \quad \wedge \quad (re^t_{i,j,i,j+1} \rightarrow rn^t_{i,j}) \quad \wedge \quad (re^t_{i,j,i,j-1} \rightarrow rn^t_{i,j})\big)$$

To prevent adjacent nodes to provide themselves reachability, acyclicity is imposed on the selected path.

$$\bigwedge_{(i,j)\in Pos\setminus Walls} \Big(\big(re^t_{i,j,i+1,j} \rightarrow (rw^t_{i,j} > rw^t_{i+1,j})\big) \quad \wedge \quad \big(re^t_{i,j,i-1,j} \rightarrow (rw^t_{i,j} > rw^t_{i-1,j})\big) \quad \wedge \quad \big(re^t_{i,j,i,j+1} \rightarrow (rw^t_{i,j} > rw^t_{i,j+1})\big) \quad \wedge \quad \big(re^t_{i,j,i,j-1} \rightarrow (rw^t_{i,j} > rw^t_{i,j-1})\big)\Big)$$

Finally, the locations where there is a ball will be forced to be not reachable:

$$\bigwedge_{(i,j)\in Pos\setminus Walls} \Big(\big(\bigvee_{k\in Balls} (b^t_k.x = i \quad \wedge \quad b^t_k.y = j) \big) \quad \rightarrow \quad \neg rn^t_{i,j} \Big)$$

Example 8.3.11 illustrates the values of the reachability variables in a particular configuration of the Willow problem.

Example 8.3.11. *Solver debug information from the Willow level using the* Reachability encoding. *The Reachable map shows the cells explored by the reachability precondition.*

```
STATE 2    Snow       Occupancy  Reachable
-#######-  ---------  -1111111-  ---------
#...1...#  -111-111-  100010001  -1110011-
#.##.##.#  -1--1--1-  101101101  -1--0--1-
#...''..#  -111--11-  100000001  -1111111-
#..#5#..#  -11-0-11-  100111001  -00-0-11-
#...p...#  -111-111-  100000001  -0111111-
##.....##  --11111--  110000011  --00101--
-#######-  ---------  -1111111-  ---------

STATE 3    Snow       Occupancy  Reachable
-#######-  ---------  -1111111-  ---------
#...1...#  -111-111-  100010001  -1110000-
#.##.##.#  -1--1--1-  101101101  -1--0--0-
#...1'..#  -111--11-  100010001  -1110000-
#..#4#..#  -11-0-11-  100111001  -11-0-00-
#...p...#  -111-111-  100000001  -1101110-
##.....##  --11111--  110000011  --11111--
-#######-  ---------  -1111111-  ---------
```

Reachable symbols description:

- **-** *no variable defined*
- **0** *false*
- **1** *true*

Example 8.3.12. *Solver solution from Tanya level using the* Reachability encoding. *Returns 17 actions which 5 are ball movement actions:*

Output	***Action***
pathfinder	*up*
solver	*right*
pathfinder	*up*
	right
solver	*down*
pathfinder	*left*
	left
	down
	down
solver	*right*
pathfinder	*down*
	right
solver	*up*
pathfinder	*left*
	left
	up
solver	*right*

8.3.4 Encoding comparisons

It is important to highlight that the *Basic encoding* and the *Cheating/Reachability encodings* do not search for the same optimality since the first encoding ensures a minimum of actions while the two other encodings ensure the minimum ball actions, but the total actions can be larger. This can be observed in the Sarah level, see Figures 10.3 and 10.4.

As seen in the encoding description, the *Cheating and Reachability encodings* are a big improvement with respect the *Basic encoding* by reducing drastically the number of time steps needed to find a solution. While the *Cheating encoding* is the fastest, for some levels, it can not find a valid plan. Meanwhile the *Reachability encoding*, though the encoded formula is larger than the *Cheating encoding*, maintains the number of time steps needed to find the solution.

Chapter 9

Deploying and testing

This chapter describers all the problems encountered and the testing methodology during the project development.

9.1 Problems

Description of the problems and difficulties found during the project development.

9.1.1 PDDL expressiveness

PDDL as it is does not support integer parameters on predicates and functions. See the following excerpt from: *pddl2.1 : An Extension to pddl for Expressing Temporal Planning Domains* (page 68):

> "Numeric expressions are not allowed to appear as terms in the language (that is, as arguments to predicates or values of action parameters). There are two justifications for this decision — a philosophical one and a pragmatic one. Philosophically we take the view that there are only a finite number of objects in the world. Numbers do not exist as unique and independent objects in the world, but only as values of attributes of objects. Our models are object-oriented in the sense that all actions can be seen as methods that apply to the objects given as their parameters. The object-oriented view does not directly inform the syntax of our representations, but is reflected in the way in which numbers are manipulated only through their relationships with the objects that are identified and named in the initial state. Pragmatically, many current planning approaches rely on being able to instantiate action schemas prior to planning, and this is only feasible if there is a finite number of action instances. The branching of the planner's search space, at choice points corresponding to action selection, is therefore always over finite ranges. The use of numeric fluent variables conflicts with this because they could occur as arguments to any predicate and would not define finite ranges.
>
> Our decision not to allow numbers to be used as arguments to actions rules out some actions that might seem intuitively reasonable. For example, an action to fly at a certain altitude might be expected to take the altitude as a number-valued argument. This is only possible in pddl2.1 if the range of numbers that can be used is finite. From a practical point of view we think that this is unlikely to be an arduous constraint and that the benefits of keeping the logical state space finite compensates for any modelling awkwardness that results."

Because it does not support integer parameters, it is not possible to encode the level as an array and query which object is in a cell. The workaround of this limitation is to use an object as a cell index, but now the problem is that it is not possible to get adjacent cells using arithmetic. This way, a new predicated is introduced which is true when two object cells are adjacent, used in the *adl* and *object-fluents* approaches.

9.1.2 Planners

The main problem with the planners is that, usually, each one implements a different subset of the language. This is the main reason why this project has three PDDL implementations, to support more planners.

Also, the latest PDDL version of the language (3.1) is practically not supported by any planner. The main reason of this is because the International Planning Competition uses only a subset of PDDL 3.1, and it is not mandatory, since they provide alternative encodings with a older subset of the language.

9.1.3 Multiple encodings

Because solving some instances in a reasonable time was impossible, two relaxed approaches had been added to the project, the *Cheating encoding* and *Reachability encoding*.

9.1.4 Class diagram

Since the code is written in Scala there is no standard UML modeling language for this language. Also because a major part of the code uses the functional paradigm, not all relations can be seen with a class diagram. If the code was pure functional, a sequence only diagram should be needed, but because Scala merges both paradigms, imperative and functional, a sequence diagram is not enough.

9.2 Testing

Description of the testing methodology used in the project development.

9.2.1 SAT Modelling

The main inconvenient of SAT modelling is testing, since there are not debugging tools available. When a formula is *unsat* or *unknown*, there is no more information about the problem. The only way to know where the error is, is to relax the formula and try to isolate the possible source.

Basically, the modelling was done by updating small parts of the formula and checking that the output did not change.

The debug information which has been used can be seen in the examples 8.3.11 and 8.3.2.

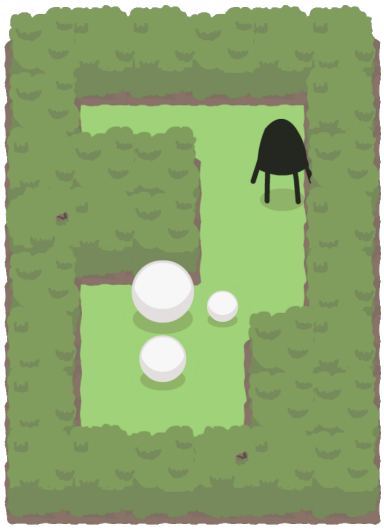
9.2.2 Validator

Every time the solver returns a plan, this is validated by the *validator*, which simulates the game. Each action will be read and the game will be modified until no more actions are available as input, or the state is invalid. The final state must have all the snowmans built to be valid.

Since the *Cheating encoding* and the *Reachability encoding* omit the character movement actions, these will be calculated offline using a pathfinding algorithm. In the case of *Cheating encoding*, when the validator returns invalid, this means that the encoder has cheated. Example see in Example 9.2.1.

Example 9.2.1. *Invalid Rebecca level solution using the Cheating encoding. Note that the solver output is correct, but the pathfinding actions are not. The algorithm returns a path because, by not finding a path in the first try, it ignores the balls. The debug information which has been used can be seen in the example 8.3.6*

Output	***Action***
pathfinder	*down*
	down
	left
	down
	left
	down
solver	*up*
pathfinder	*right*
	up
	right
solver	*left*



Chapter 10

Results

10.1 Snowman editor

The *Snowman editor* (Figure 10.1) is a complete tool which facilitates the level design by allowing the user to create a new level, try it in-game and also find a solution. The tool has multiple solvers which use the SMT approach but also allows to generate PDDL problem instances which can be used externally with the desired planner. All the functionalities are described in the *User Manual* section.

10.2 Experimental results

Only the levels present in the final game are included in the results. The results has been computed using the following solver configuration:

Start time step auto

Max time steps 1000

Timeout (s) 3600

Threads 1

Invariant ball sizes false

and with the hardware and software:

OS Debian 9

CPU i7-6700K stock configuration

RAM 16GB DDR4 at 3200MHz

Yices 2 version 2.5.4

Previous results have shown that, in general, enabling the *Ball sizes* invariant gets worse results than without it. By this reason, all the tests have been calculated without it.

The *Basic encoding* only has been able to solve 9 of the 30 tested levels. While there are unsolved levels with a smaller plan than the longest plan solved, these levels have a larger playable area, hence a bigger space search space.

Meanwhile, the Cheating encoding has much better results with 27 of the 30 levels solved. Also, it is important to notice that the levels with one snowman[1] have much lower times than the *Basic encoding* times. It is interesting because this encoding also has been able to solve a level with two snowmen (*Jill & Jack*).

It is important to highlight that not all the plans from the *Cheating encoding* are invalid plans. The encoding has been able to give 11 valid plans of 25 solved levels.

Finally the *Reachability encoding* results are very noticeable since, even with the reachability precondition, it has been able to solve the same levels as the Cheating encoding.

In the last two encodings doesn't seem to have any trouble solving the single snowman levels, but when there are more than one snowman the times increase so much that the solver gives timeout. As seen in the

[1] A level name is the names of the snowmen

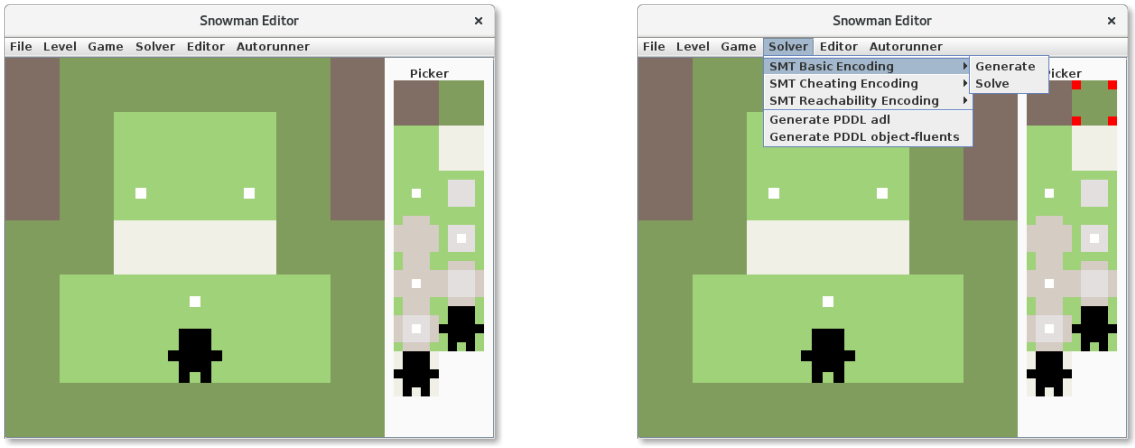


Figure 10.1: *Snowman Editor* with the level Sarah loaded. The tool is composed by a level representation (right), a tile type picker (right) and a menu bar (top) which allow to use all the tool functionalities.

encoding, this is caused by the number of actions and, because these are more complex levels, their size is also bigger. The number of actions n given the number of balls b is given by the formula:

Basic encoding $n = 4(charactermovement) * b * 4(directions)$

Cheating and Reachability encodings $n = b * 4(directions)$

Is interesting to highlight that the *Reachability encoding*, because the reachability predicate can reduce the space search in front of the *Cheating encoding*. This can be seen in the Alex level in the Figure 10.4.

Another interesting case is when the *Cheating encoding* finds smaller, and also valid, plans than the *Reachability encoding*. This is because although they share the same number of ball actions, the order of these is not the same and when reconstructing this state differences may produce a longer path.

Because a bug was found in this document revision, which affects all tree encodings, because of time limitations, all the results with a timeout are outdated. A preliminary result showed that, by removing the bug, the performance has increased. This means that all the timeout solutions have been able to solve more time steps before reaching the timeout.

10.3 Normative and legislation

This project has no problems with the current legislation and normative.

- Refering the *LOPDE* and *GDPR*, this project does not store any personal data.
- Refering to the *LSSICE*, this project is not an economic activity.

Also this project can be seen as a mod, but the editor does not include any game files and the game is needed to use all the editor functionalities. It could be argued if the editor generates a deviated work but, since this deviated work would not be distributed and would be seen as a fair use, no copyright law would be infringed.

```
Timesteps: 5 sat: false Time: 501 TotalTime: 527 (ms)
Timesteps: 6 sat: false Time: 276 TotalTime: 804 (ms)
Timesteps: 7 sat: false Time: 274 TotalTime: 1078 (ms)
Timesteps: 8 sat: true Time: 441 TotalTime: 1601 (ms)

Solved: true
Valid: true
Actions: 26
    up
    right
    up
    left
    up
    up
    right
    down
    down
    left
    down
    down
    right
    right
    up
    left
    left
    down
    left
    up
    right
    up
    up
    up
    left
    down
Ball references from initial state:
    Ball (0): Location(Coordinate(2,4),SmallBall)
    Ball (1): Location(Coordinate(3,2),SmallBall)
    Ball (2): Location(Coordinate(4,4),SmallBall)
Ball Actions: 8
    BallUp(1)
    BallLeft(1)
    BallDown(2)
    BallDown(2)
    BallLeft(2)
    BallLeft(2)
    BallUp(2)
    BallDown(0)
```

Figure 10.2: Solver output from the Sharah level using the *Reachability encoding*. Because the *Ball Actions* are applied to a ball, each ball has an identifier. *Ball referenes from initial state* has the information to match each ball identifier with the ingame ball.

		Human		Basic Encoding					
Level	**Map size**	**Actions**	**B. Actions**[a]	**Solved**	**Steps**[b]	**Time**	**Valid**	**Actions**	**B. Actions**
Lucy	24	19	8	TRUE	19	10937	TRUE	19	8
Andy	25	19	6	TRUE	19	44140	TRUE	19	6
Tanya	15	18	6	TRUE	17	28474	TRUE	17	5
Kevin	24	42	11	TRUE	38	1827178	TRUE	38	11
Sarha	19	26	8	TRUE	26	125156	TRUE	26	8
Louise	14	52	15	TRUE	33	1356625	TRUE	33	13
Alice	35	53	24	FALSE	36	3600001			
Freya	19	49	15	FALSE	33	3600001			
Chris	16	31	7	TRUE	31	243521	TRUE	31	7
Alex	23	56	13	FALSE	26	3600001			
Julian	35	44	13	FALSE	35	3600009			
Ryan	24	52	15	FALSE	31	3600001			
Sally	25	92	15	FALSE	35	3600000			
Lauren	16	66	15	FALSE	33	3600001			
Jill & Jack	22	56	26	FALSE	25	3600001			
Willow	34	95	14	FALSE	33	3600001			
Ben & Alan	32	177	40	FALSE	26	3600001			
Adam	21	97	12	FALSE	30	3600001			
Rob, James & Matthew	38	152	47	FALSE	20	3600001			
Jessica & Amelia	25	84	26	FALSE	21	3600001			
Cynthia & Michael	34	83	34	FALSE	28	3600001			
David	40	24	7	TRUE	23	93841	TRUE	23	8
Lydia	21	27	7	TRUE	27	129804	TRUE	27	7
Mary	14	46	10	FALSE	35	3600001			
Helen	24	57	13	FALSE	30	3600001			
Paul	25	67	28	FALSE	37	3600001			
Kate	25	36	11	FALSE	30	3600000			
Rebecca	18	27	6	TRUE	24	119374	TRUE	24	6
William	25	50	15	FALSE	36	3600001			
Zoe & Richard	28	92	29	FALSE	25	3600001			

Figure 10.3: Human and Basic encoding results

[a]Ball Actions
[b]Time steps

	Cheating Encoding						Reachability Encoding					
Level	**Solved**	**Steps**	**Time**	**Valid**	**Actions**	**B. Actions**	**Solved**	**Steps**	**Time**	**Valid**	**Actions**	**B. Actions**
Lucy	TRUE	8	469	TRUE	19	8	TRUE	8	855	TRUE	23	8
Andy	TRUE	6	356	TRUE	23	6	TRUE	6	525	TRUE	27	6
Tanya	TRUE	5	168	TRUE	21	5	TRUE	5	247	TRUE	17	5
Kevin	TRUE	11	4865	TRUE	38	11	TRUE	11	6674	TRUE	40	11
Sarha	TRUE	8	493	FALSE	36	8	TRUE	8	621	TRUE	34	8
Louise	TRUE	7	439	FALSE	27	7	TRUE	13	29818	TRUE	35	13
Alice	TRUE	11	14893	FALSE	38	11	TRUE	19	1172751	TRUE	66	19
Freya	TRUE	5	194	FALSE	22	5	TRUE	13	11829	TRUE	46	13
Chris	TRUE	7	346	TRUE	31	7	TRUE	7	507	TRUE	31	7
Alex	TRUE	13	69187	TRUE	70	13	TRUE	13	77492	TRUE	67	13
Julian	TRUE	11	5122	FALSE	44	11	TRUE	13	44354	TRUE	73	13
Ryan	TRUE	9	1979	FALSE	36	9	TRUE	15	109487	TRUE	60	15
Sally	TRUE	13	16963	TRUE	71	13	TRUE	13	24129	TRUE	71	13
Lauren	TRUE	9	2488	FALSE	34	9	TRUE	11	7104	TRUE	42	11
Jill & Jack	TRUE	14	135360	FALSE	49	14	TRUE	16	472872	TRUE	60	16
Willow	TRUE	14	96567	FALSE	101	14	TRUE	14	87312	TRUE	95	14
B. & A.	FALSE	12	3600001				FALSE	14	3600001			
Adam	TRUE	12	10069	TRUE	79	12	TRUE	12	9280	TRUE	89	12
R., J. & M.	FALSE	8	3600001				FALSE	8	3600000			
J. & A.	FALSE	12	3600001				FALSE	12	3600001			
C. & M.	FALSE	11	3600001				FALSE	12	3600001			
David	TRUE	7	796	TRUE	24	7	TRUE	7	2324	TRUE	32	7
Lydia	TRUE	7	430	TRUE	27	7	TRUE	7	503	TRUE	27	7
Mary	TRUE	10	1455	TRUE	45	10	TRUE	10	1571	TRUE	41	10
Helen	TRUE	11	8748	TRUE	52	11	TRUE	11	11899	TRUE	58	11
Paul	TRUE	9	1203	FALSE	40	9	TRUE	26	216808	TRUE	72	26
Kate	TRUE	10	7868	TRUE	53	10	TRUE	10	6400	TRUE	37	10
Rebecca	TRUE	2	34	FALSE	11	2	TRUE	6	255	TRUE	24	6
William	TRUE	13	9434	FALSE	67	13	TRUE	15	14590	TRUE	57	15
Z. & R.	FALSE	10	3600001				FALSE	11	3600002			

Figure 10.4: Cheating and reachability encoding results

Chapter 11

Conclusions

The objective of this project was to give a tool which helped the design of *A Good Snowman Is Hard To Build* levels. This objective has been accomplished creating a complete tool with 5 solving methods. Also the tool allows to try the level in-game, which is an important aspect of a game-developing tool.

It is important to note that, despite the hardness of the problem, three of these solving methods are "optimal". This hardness is the main reason why some levels have not been yet solved, but given enough time, the tool could be able to solve them. In any case, we remark the good results obtained with the *Reachability encoding*, which is able to provide valid plans in minutes and even, in some cases, in seconds, for all one snowman instances.

Also remarkable is the extension that has been made to the SMT encoding of the *s-t-reachability* problem by being able to set the source and target in solving-time instead of encoding-time, as well as dealing with dynamic graphs (since the positions of the balls are not known when building the encoding).

On the other hand, the PDDL approach, given all the problems with the planners, is not closed. There is still work to be done like searching newer planners, or getting the results by running all the problem instances.

This project has used multiple pieces of knowledge acquired in the degree, specially in the last year, in the computational branch. Some knowledge used is:

- SAT and SMT: Declarative programming subject (Programació declarativa. Aplicacions)
- Context-free grammar: Compiler subject and Computational principles subject (Fonaments de computació)
- Functional programming and Scala: Programming paradigms subject (Paradigmes i llenguatges de programació)
- Planning problems: Artificial intelligence subject (Intel·ligència artificial)
- Programming patterns: Software engineering subject (Enginyeria del software)

This project has provided a good knowledge on planning problems and how to approach them. Also has teach how to search and understand state of the art articles and the basic guidelines to create them.

Chapter 12

Further work

This chapter describes all the further work which could be done in base this project.

12.1 GUI Implementation

There are some parts in the Snowman Editor which are still implemented in terminal-like style. These should be integrated into the GUI. Parts which are currently not implemented:

- SMT Solver solution visualization and statistics
- SMT Solver solution debug information
- Validator module visual feedback

12.2 Numeric-fluents

Since most of the planners do not support the latest *PDDL* subset *object-fluents*, the *Snowman problem* can be described using a different *PDDL* subset *numericfluents*.

The main idea of this approach is to define each entity, character, balls and snow, by numeric coordinates. This avoids using the *next* predicate and the *location* parameters used in the *adl* approach.

12.3 SAT

It is unclear if a SAT encoding could be faster than the current SMT encoding. A possible future work could be to implement a *translator* from the *internal language* to SAT and the use of a SAT solver.

12.4 Optimization

During the different iterations of the solver *SMT* implementation, a large number of optimizations have been implemented, but still there are some which have not.

These optimizations are marked all along the code with the *OPTIMIZATION* tag to ease its later localization. Some possible optimizations are:

- NO_WALL_IN_FRONT could be encoded using arithmetic
- UPDATE_BALL_SIZE and UPDATE_SNOW_VARIABLES could be merged
- NO_WALL_IN_FRONT could check for walls instead of playable locations, depending on the shortest clause
- Merge occupancy variables and reachability nodes in the *Reachability encoding*

12.5 More invariants

A way to improve current solving times is by adding invariants. Not all invariant may improve the times, so they have to be tested. Some possible invariants are:

- Some ball positions are invalid (for instance, when building just one snowman, if two ball are located in two distinct corners the problem has no solution).
- The distance between snow balls has to be smaller than the remaining timesteps in the current encoding.
- The number of snow cells remaining must be larger or equal than the ball size changes needed to form the snowmans.
- Some ball positions create dead locks between them.

12.6 Level designs given a certain numbers of steps

Because of the nature of SAT, there is no start nor end regarding plan search. This allows to make a query backwards, given a goal state and a number of timesteps, find the starting state. This approach can provide a powerful tool on level design since, instead of solving instances, it is generating them. However, this approach is not straightforward at all to implement.

12.7 Other games

Initially the project was thought to use the *Hitman Go* as the concrete game. The game was discarded due to its much more complex game mechanics. Once done this project and with a larger vision on all the topics needed to resolve this kind of problems, the previous discarded game seems more plausible and could result in even better results. Here follow some differences between them:

- The levels in the Go games from *Square Enix* are much smaller because there are predefined paths instead of a grid, which reduces the search depth.
- Although there are more actions to perform in the Go games, in every state, only a few actions can be performed (approximately no more than 6). Also, because each level is focused in a few game mechanics, only a few actions are present int each level, which can reduce the ramification.
- At first it was unclear how to encode the enemies agents, but with more knowledge it seams possible to encode the enemies actions in the encoding effect statement. Also these agents follow predefined rules which makes the codification relatively easy.

Some future work could be the encoding these *Go* games from *Square Enix* like the *Hitman Go*, or any other turn-based game.

Chapter 13

Bibliography

[1] Malik Ghallab, Craig Knoblock, David Wilkins, Anthony Barrett, Dave Christianson, Marc Friedman, Chung Kwok, Keith Golden, Scott Penberthy, David Smith, Ying Sun, and Daniel Weld. Pddl - the planning domain definition language. 08 1998.

[2] Maria Fox and Derek Long. Pddl2.1: An extension to pddl for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20(1):61–124, December 2003.

[3] Stefan Edelkamp. Pddl2.2: The language for the classical part of the 4th international planning competition. 01 2004.

[4] Alfonso Gerevini and Derek Long. Bnf description of pddl3.0. 10 2005.

[5] Patrik Haslum. Changes in pddl 3.1. `http://icaps-conference.org/ipc2008/deterministic/PddlExtension.html`, 9 2011. Accessed:2018-9-28.

[6] Daniel L. Kovacs. Complete bnf description of pddl 3.1 (completely corrected). 2011.

[7] Icaps competitions. `http://www.icaps-conference.org/index.php/Main/Competitions`. Accessed:2018-9-28.

[8] Wikipedia contributors. Strips — Wikipedia, the free encyclopedia. `https://en.wikipedia.org/w/index.php?title=STRIPS&oldid=776534185`, 2017. [Online; accessed 1-September-2018].

[9] Clark Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. *Satisfiability modulo theories*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. 1 edition, 2009.

[10] Miquel Bofill, Joan Espasa, and Mateu Villaret. The RANTANPLAN planner: system description. *Knowledge Eng. Review*, 31(5):452–464, 2016.

[11] Bruno Dutertre. Yices 2.2. In *Proceedings of the 26th International Conference on Computer Aided Verification, CAV*, volume 8559 of *Lecture Notes in Computer Science*, pages 737–744, 2014.

[12] SRI International. The yices smt solver. `http://yices.csl.sri.com/`, May 2018. Accessed:2018-9-28.

[13] Malte Helmert. The fast downward planning system. *Journar of Artificial Intlligence Research*, 26(1):191–246, July 2006.

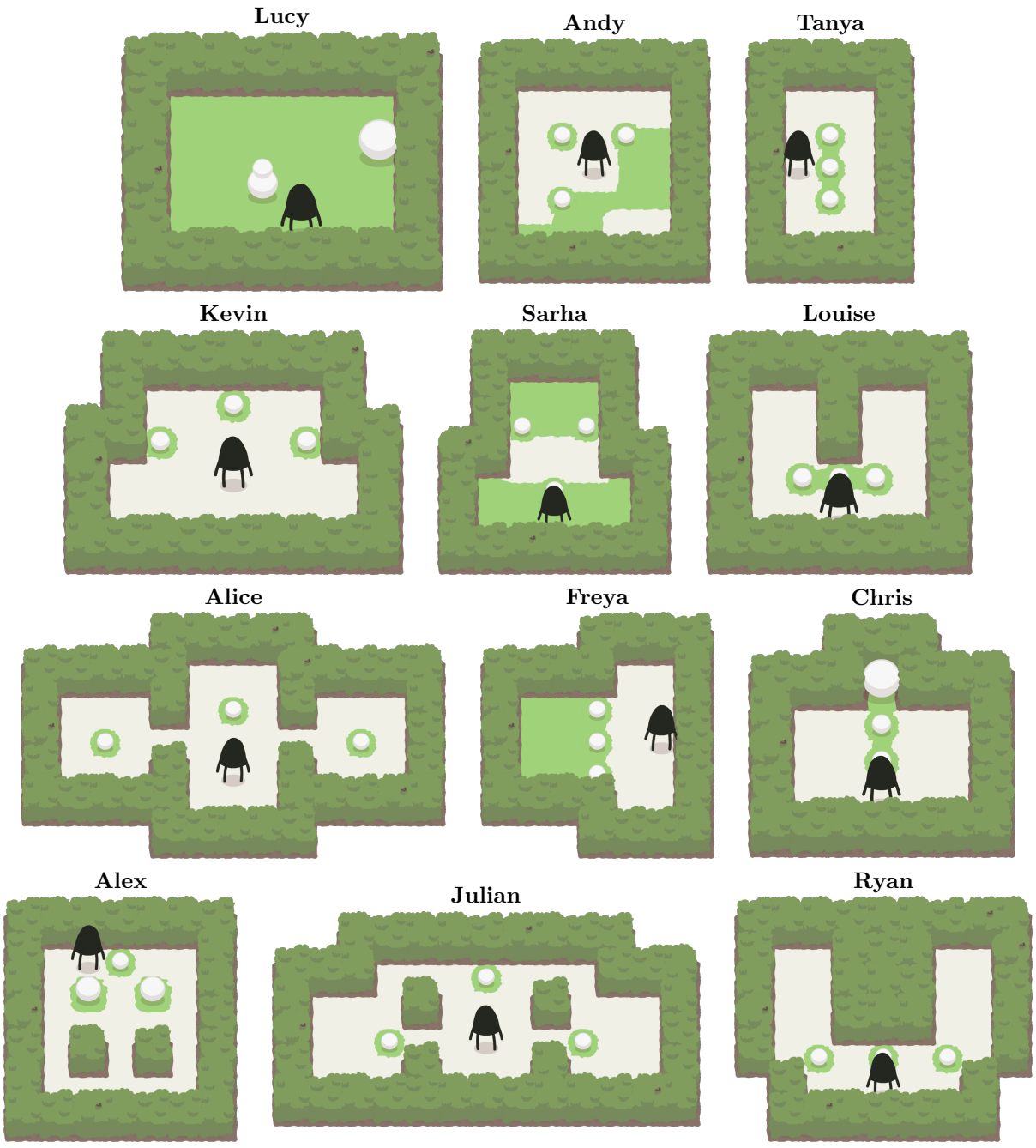
[14] Martin Gebser, Tomi Janhunen, and Jussi Rintanen. SAT modulo graphs: Acyclicity. In *Proceedings of the 14th European Conference on Logics in Artificial Intelligence - JELIA*, volume 8761 of *Lecture Notes in Computer Science*, pages 137–151, 2014.

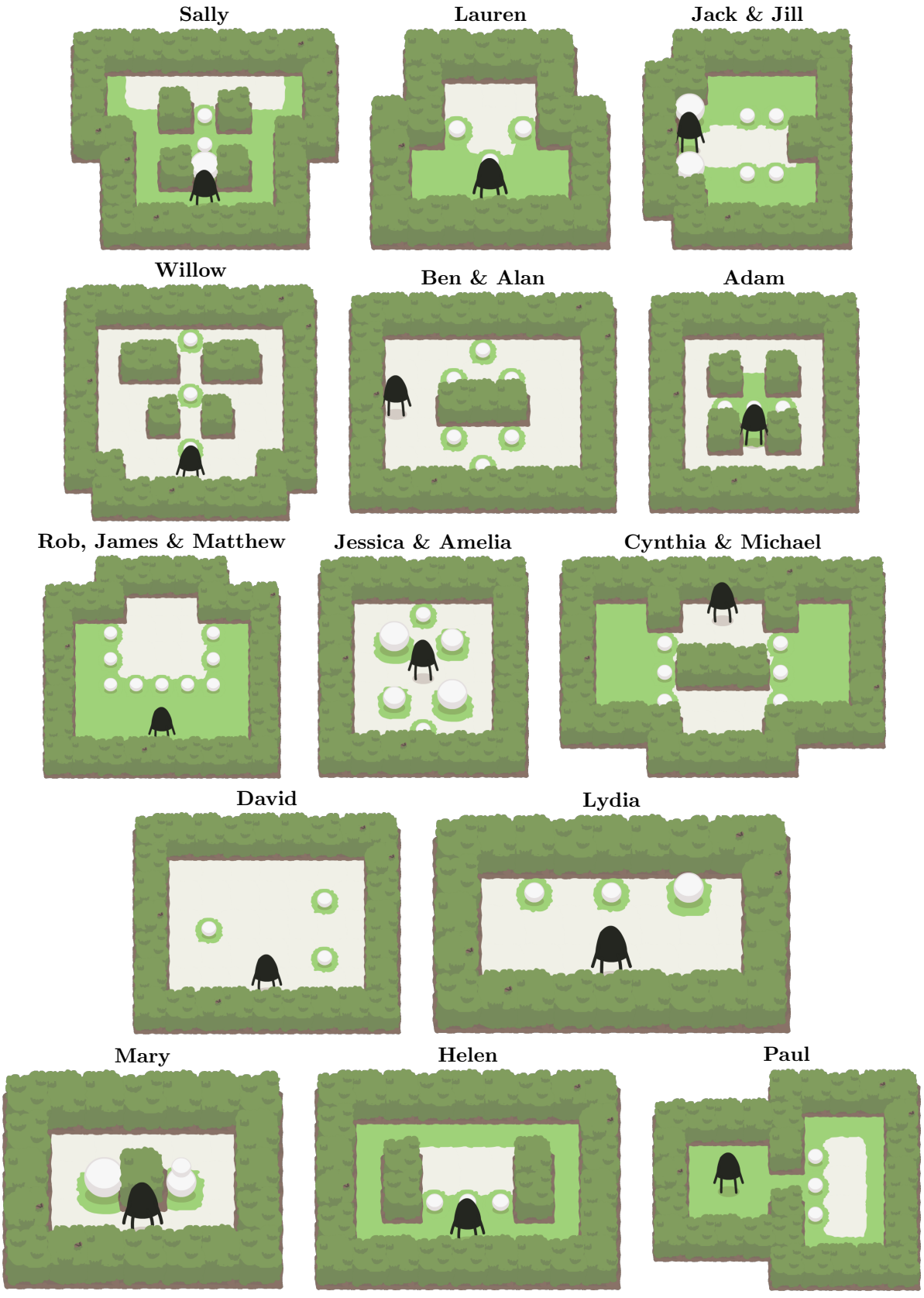
Chapter 14

Appendix

14.1 A Good Snowman Is Hard To Build levels

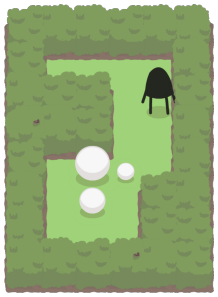
All the levels from the *A Good Snowman Is Hard To Build* game found in the "real world".



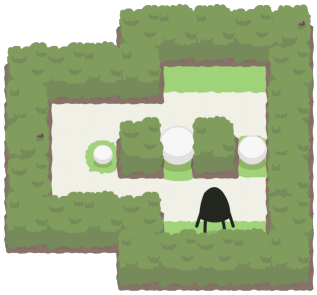

Sally
Lauren
Jack & Jill
Willow
Ben & Alan
Adam
Rob, James & Matthew
Jessica & Amelia
Cynthia & Michael
David
Lydia
Mary
Helen
Paul

Kate

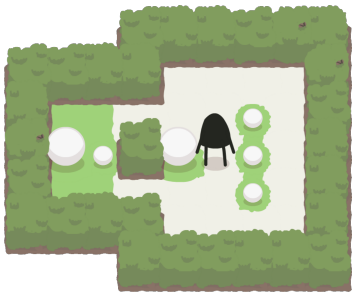
Rebecca



William



Zoe & Richard



14.2 *adl* domain

```
(define (domain snowman-adl)

    (:requirements
        :typing
        :negative-preconditions
        :equality
        :disjunctive-preconditions
        :conditional-effects
    )

    (:types
        location direction ball size - object
    )

    (:predicates
        (snow ?l - location)
        (next ?from ?to - location ?dir - direction)
        (occupancy ?l - location)
        (character-at ?l - location)
        (ball-at ?b - ball ?l - location)
        (ball-size-small ?b - ball)
        (ball-size-medium ?b - ball)
        (ball-size-large ?b - ball)
        (goal)
    )

    (:action move_character

     :parameters
       (?from ?to - location ?dir - direction)

     :precondition
        (and
            (next ?from ?to ?dir)
            (character-at ?from)
            (not (occupancy ?to)))

     :effect
        (and
            (not (character-at ?from))
            (character-at ?to))
    )

    (:action move_ball

     :parameters
        (?b - ball ?ppos ?from ?to - location ?dir - direction)

     :precondition
```

```
  (and
      (next ?ppos ?from ?dir)
      (next ?from ?to ?dir)
      (ball-at ?b ?from)
      (character-at ?ppos)
      (forall (?o - ball)
          (or
              (= ?o ?b)
              (or
                  (not (ball-at ?o ?from))
                  (or
                      (and
                          (ball-size-small ?b)
                          (ball-size-medium ?o))
                      (and
                          (ball-size-small ?b)
                          (ball-size-large ?o))
                      (and
                          (ball-size-medium ?b)
                          (ball-size-large ?o))))))
      (or
          (forall (?o - ball)
              (or
                  (= ?o ?b)
                  (not (ball-at ?o ?from))))
          (forall (?o - ball)
                  (not (ball-at ?o ?to))))
      (forall (?o - ball)
              (or
                  (not (ball-at ?o ?to))
                  (or
                      (and
                          (ball-size-small ?b)
                          (ball-size-medium ?o))
                      (and
                          (ball-size-small ?b)
                          (ball-size-large ?o))
                      (and
                          (ball-size-medium ?b)
                          (ball-size-large ?o))))))
:effect
  (and
      (when
          (forall (?o - ball)
              (or (= ?o ?b)
              (ball-at ?o ?to)))
          (goal))
      (not (occupancy ?from))
      (occupancy ?to)
      (not (ball-at ?b ?from))
      (ball-at ?b ?to)
      (when
```

```
            (forall (?o - ball)
                (or
                    (= ?o ?b)
                    (not (ball-at ?o ?from))))
            (and
                (not (character-at ?ppos))
                (character-at ?from)))
        (not (snow ?to))
        (when
            (and
                (snow ?to)
                (ball-size-small ?b))
            (and
                (not (ball-size-small ?b))
                (ball-size-medium ?b)))
        (when
            (and
                (snow ?to)
                (ball-size-medium ?b))
            (and
                (not (ball-size-medium ?b))
                (ball-size-large ?b))))
    )
)
```

14.3 *object-fluents* domain

```
(define (domain snowman-object-fluents)

    (:requirements
        :typing
        :negative-preconditions
        :equality
        :disjunctive-preconditions
        :conditional-effects
        :object-fluets
    )

    (:types
        location ball - object
    )

    (:predicates
        (snow ?l - location)
        (occupancy ?l - location)
        (goal)
    )

    (:functions
        (character-at) - location
        (ball-at ?b - ball) - location
        (next-up ?l - location) - location
        (next-down ?l - location) - location
        (next-right ?l - location) - location
        (next-left ?l - location) - location
    )

    (:action move_up_character

     :parameters

     :precondition
        (not (occupancy (next-up (character-at))))

     :effect
        (assign (character-at) (next-up (character-at)))
    )

    (:action move_down_character

     :parameters

     :precondition
        (not (occupancy (next-down (character-at))))

     :effect
```

```
    (assign (character-at) (next-down (character-at)))
)

(:action move_right_character

 :parameters

 :precondition
    (not (occupancy (next-right (character-at))))

 :effect
    (assign (character-at) (next-right (character-at)))
)

(:action move_left_character

 :parameters

 :precondition
    (not (occupancy (next-left (character-at))))

 :effect
    (assign (character-at) (next-left (character-at)))
)

(:action move_up_ball

 :parameters
    (?b - ball)

 :precondition
    (and
        (= (character-at) (next-down (ball-at ?b)))
        (forall (?o - ball)
            (or
                (= ?o ?b)
                (or
                    (not (= (ball-at ?o) (ball-at ?b)))
                    (or
                        (and
                            (ball-size-small ?b)
                            (ball-size-medium ?o))
                        (and
                            (ball-size-small ?b)
                            (ball-size-large ?o))
                        (and
                            (ball-size-medium ?b)
                            (ball-size-large ?o))))))
        (or
            (forall (?o - ball)
                (or
                    (= ?o ?b)
```

```
                    (not (= (ball-at ?o) ((ball-at ?b))))))
            (forall (?o - ball)
                    (not (= (ball-at ?o) (next-up (ball-at ?b))))))
        (forall (?o - ball)
                (or
                    (not (= (ball-at ?o) (next-up (ball-at ?b))))
                    (or
                        (and
                            (ball-size-small ?b)
                            (ball-size-medium ?o))
                        (and
                            (ball-size-small ?b)
                            (ball-size-large ?o))
                        (and
                            (ball-size-medium ?b)
                            (ball-size-large ?o))))))

  :effect
    (and
        (when
            (forall (?o - ball)
                (or (= ?o ?b)
                (= (ball-at ?o) (next-up (ball-at ?b)))))
            (goal))
        (not (occupancy (ball-at ?b)))
        (occupancy (next-up (ball-at ?b)))
        (assign (ball-at ?b) (next-up (ball-at ?b)))
        (when
            (forall (?o - ball)
                (or
                    (= ?o ?b)
                    (not (= (ball-at ?o) (ball-at ?b)))))
            (assign (character-at) (ball-at ?b)))
        (not (snow (next-up (ball-at ?b))))
        (when
            (and
                (snow (next-up (ball-at ?b)))
                (ball-size-small ?b))
            (and
                (not (ball-size-small ?b))
                (ball-size-medium ?b)))
        (when
            (and
                (snow (next-up (ball-at ?b)))
                (ball-size-medium ?b))
            (and
                (not (ball-size-medium ?b))
                (ball-size-large ?b))))
)

(:action move_down_ball
```

```
:parameters
   (?b - ball)

:precondition
   (and
       (= (character-at) (next-up (ball-at ?b)))
       (forall (?o - ball)
           (or
               (= ?o ?b)
               (or
                   (not (= (ball-at ?o) (ball-at ?b)))
                   (or
                       (and
                           (ball-size-small ?b)
                           (ball-size-medium ?o))
                       (and
                           (ball-size-small ?b)
                           (ball-size-large ?o))
                       (and
                           (ball-size-medium ?b)
                           (ball-size-large ?o))))))
       (or
           (forall (?o - ball)
               (or
                   (= ?o ?b)
                   (not (= (ball-at ?o) ((ball-at ?b))))))
           (forall (?o - ball)
                   (not (= (ball-at ?o) (next-down (ball-at ?b))))))
       (forall (?o - ball)
               (or
                   (not (= (ball-at ?o) (next-down (ball-at ?b))))
                   (or
                       (and
                           (ball-size-small ?b)
                           (ball-size-medium ?o))
                       (and
                           (ball-size-small ?b)
                           (ball-size-large ?o))
                       (and
                           (ball-size-medium ?b)
                           (ball-size-large ?o))))))

 :effect
   (and
       (when
           (forall (?o - ball)
               (or (= ?o ?b)
               (= (ball-at ?o) (next-down (ball-at ?b)))))
           (goal))
       (not (occupancy (ball-at ?b)))
       (occupancy (next-down (ball-at ?b)))
       (assign (ball-at ?b) (next-down (ball-at ?b)))
```

```
        (when
            (forall (?o - ball)
                (or
                    (= ?o ?b)
                    (not (= (ball-at ?o) (ball-at ?b)))))
            (assign (character-at) (ball-at ?b)))
        (not (snow (next-down (ball-at ?b))))
        (when
            (and
                (snow (next-down (ball-at ?b)))
                (ball-size-small ?b))
            (and
                (not (ball-size-small ?b))
                (ball-size-medium ?b)))
        (when
            (and
                (snow (next-down (ball-at ?b)))
                (ball-size-medium ?b))
            (and
                (not (ball-size-medium ?b))
                (ball-size-large ?b))))
)

(:action move_right_ball

 :parameters
    (?b - ball)

 :precondition
    (and
        (= (character-at) (next-left (ball-at ?b)))
        (forall (?o - ball)
            (or
                (= ?o ?b)
                (or
                    (not (= (ball-at ?o) (ball-at ?b)))
                    (or
                        (and
                            (ball-size-small ?b)
                            (ball-size-medium ?o))
                        (and
                            (ball-size-small ?b)
                            (ball-size-large ?o))
                        (and
                            (ball-size-medium ?b)
                            (ball-size-large ?o))))))
        (or
            (forall (?o - ball)
                (or
                    (= ?o ?b)
                    (not (= (ball-at ?o) ((ball-at ?b))))))
            (forall (?o - ball)
```

```
                (not (= (ball-at ?o) (next-right (ball-at ?b))))))
        (forall (?o - ball)
                (or
                    (not (= (ball-at ?o) (next-right (ball-at ?b))))
                    (or
                        (and
                            (ball-size-small ?b)
                            (ball-size-medium ?o))
                        (and
                            (ball-size-small ?b)
                            (ball-size-large ?o))
                        (and
                            (ball-size-medium ?b)
                            (ball-size-large ?o))))))

  :effect
    (and
        (when
            (forall (?o - ball)
                (or (= ?o ?b)
                (= (ball-at ?o (next-right (ball-at ?b))))))
            (goal))
        (not (occupancy (ball-at ?b)))
        (occupancy (next-right (ball-at ?b)))
        (assign (ball-at ?b) (next-right (ball-at ?b)))
        (when
            (forall (?o - ball)
                (or
                    (= ?o ?b)
                    (not (= (ball-at ?o) (ball-at ?b)))))
            (assign (character-at) (ball-at ?b)))
        (not (snow (next-right (ball-at ?b))))
        (when
            (and
                (snow (next-right (ball-at ?b)))
                (ball-size-small ?b))
            (and
                (not (ball-size-small ?b))
                (ball-size-medium ?b)))
        (when
            (and
                (snow (next-right (ball-at ?b)))
                (ball-size-medium ?b))
            (and
                (not (ball-size-medium ?b))
                (ball-size-large ?b))))
)

(:action move_left_ball

 :parameters
    (?b - ball)
```

```
:precondition
   (and
       (= (character-at) (next-right (ball-at ?b)))
       (forall (?o - ball)
           (or
               (= ?o ?b)
               (or
                   (not (= (ball-at ?o) (ball-at ?b)))
                   (or
                       (and
                           (ball-size-small ?b)
                           (ball-size-medium ?o))
                       (and
                           (ball-size-small ?b)
                           (ball-size-large ?o))
                       (and
                           (ball-size-medium ?b)
                           (ball-size-large ?o))))))
       (or
           (forall (?o - ball)
               (or
                   (= ?o ?b)
                   (not (= (ball-at ?o) ((ball-at ?b))))))
           (forall (?o - ball)
                   (not (= (ball-at ?o) (next-left (ball-at ?b))))))
       (forall (?o - ball)
               (or
                   (not (= (ball-at ?o) (next-left (ball-at ?b))))
                   (or
                       (and
                           (ball-size-small ?b)
                           (ball-size-medium ?o))
                       (and
                           (ball-size-small ?b)
                           (ball-size-large ?o))
                       (and
                           (ball-size-medium ?b)
                           (ball-size-large ?o))))))

 :effect
   (and
       (when
           (forall (?o - ball)
               (or (= ?o ?b)
               (= (ball-at ?o) (next-left (ball-at ?b)))))
           (goal))
       (not (occupancy (ball-at ?b)))
       (occupancy (next-left (ball-at ?b)))
       (assign (ball-at ?b) (next-left (ball-at ?b)))
       (when
           (forall (?o - ball)
```

```
                (or
                    (= ?o ?b)
                    (not (= (ball-at ?o) (ball-at ?b)))))
            (assign (character-at) (ball-at ?b)))
        (not (snow (next-left (ball-at ?b))))
        (when
            (and
                (snow (next-left (ball-at ?b)))
                (ball-size-small ?b))
            (and
                (not (ball-size-small ?b))
                (ball-size-medium ?b)))
        (when
            (and
                (snow (next-left (ball-at ?b)))
                (ball-size-medium ?b))
            (and
                (not (ball-size-medium ?b))
                (ball-size-large ?b))))
    )
)
```

Chapter 15

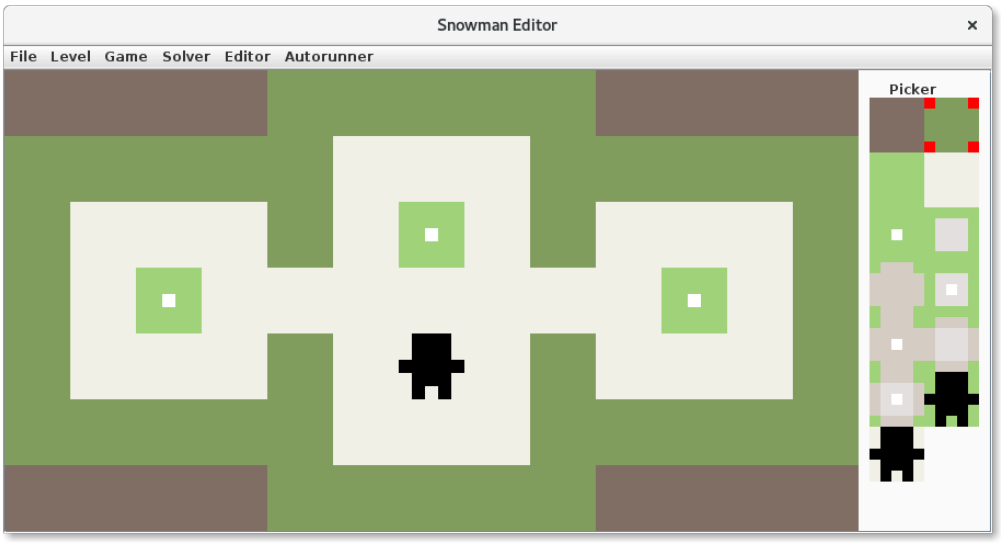
User manual

15.1 Application requirements

To be able to perform all the *Snowman Editor* actions the game *A Good Snowman Is Hard To Build* and the solver *Yices 2* must be installed the computer. The paths where are installed must be denoted in the file *snowman_editor.config*.

If the *snowman_editor.config* file is not included between the *Snowman Editor* files, it will be generated in the first run.

If a requirement is not installed, the functionalities related to this requirement will be disabled.



15.2 Functionalities description

This section describes all the actions can be made with the *Snowman Editor*.

15.2.1 Picker layout

Allows to select a tile type to change the level layout, like a painting application.

15.2.2 File menu

Level file related functionalities.

15.2.2.1 New

Allows to create a new level defining its size.

15.2.2.2 Open

Open a level file.

15.2.2.3 Save

Save a game file. A level file contents does not include the level name.

15.2.2.4 Load

The game is needed to use these functionalities. It allows to load any level defined in the game resources. Note that the resources have more levels than the actual game uses. The levels which the game does not use are the following:

- Unused
- Unused (two)
- Ruth
- Kate, Garry & Craig
- Claire
- Zoe 2 & Richard 2
- Unnamed
- Hard, Harder & Hardest
- One, Two & Three
- A & B

15.2.3 Level menu

Level editor related functionalities.

15.2.3.1 Info

Allows to get different informations of the current level such as its size, number of balls, different tile counts, etc.

15.2.3.2 Validate

Validates if a level can be playable. Must satisfy the following restrictions:

- The level must have character
- The number of balls must be multiple of 3 and bigger than 0.

15.2.4 Game menu

Mod related functionalities. The game is needed to use these functionalities.

15.2.4.1 Load and Run

Loads the level and runs the game. If the game uses the original files, it will save them to a new created directory *game_backup*.

15.2.4.2 Run

Runs the game without modifying any file. If a custom level is loaded it will remain loaded.

15.2.4.3 Restore

Restores the game to the previous state with the *game_backup* files.

15.2.5 Solver menu

Solver-related functionalities. The *Yices 2* solver is needed to use these functionalities.

The SMT options also allow to solve or generate a single encoding given the timesteps number, while the PDDL options only allow to generate the problem files. The domain files for the PDDL can be found among the *Snowman Editor* files.

The possible SMT solving options are:

Start time step Time step lower bound. *auto* for a calculated lower bound, or a positive number.

Max time steps Maximum number of time steps allowed before the execution is terminated.

Timeout (s) Timeout in seconds before the execution is terminated.

Threads Number of threads the solver can use. Each thread will be used for launching a different time step encoding.

Invariants Enable or disable specific invariants.

15.2.5.1 SMT Basic encoding

Solves the level by running the *Yices 2* solver with the Basic encoding.

15.2.5.2 SMT Cheating encoding

Solves the level by running the *Yices 2* solver with the Cheating encoding.

15.2.5.3 SMT Reachability encoding

Solves the level by running the *Yices 2* solver with the Reachability encoding.

15.2.5.4 Generate PDDL adl

Generates a PDDL problem file for the domain *adl*.

15.2.5.5 Generate PDDL object-fluents

Generates a PDDL problem file for the domain *object-fluents*.

15.2.6 Editor menu

Level editor display related options.

15.2.6.1 Coordinates

Displays the coordinates on each tile. Useful for debugging.

15.2.6.2 Mateu mode

Enable or disable Mateu mode.

15.2.7 Autorunner menu

Yices 2 solver is needed to use these functionalities. Allows to run a solver to multiple levels files located in a directory. An output directory will be required to save all the results.

15.2.7.1 Solve SMT Basic

Run the autorunner with the *Basic encoding*.

15.2.7.2 Solve SMT Cheating

Run the autorunner with the *Cheating encoding*.

15.2.7.3 Solve SMT Reachability

Run the autorunner with the *Reachability encoding*.