Electronics and Computer Science
Faculty of Physical Sciences and Engineering
University of Southampton

Ross Viljoen
Monday 30th November, 2020

A Tool for Symbolic Resource-Aware Reactive Synthesis

Project supervisor: Dr. Corina Cîrstea
Second Examiner: Dr. Enrico Gerding

A project report submitted for the award of
BSc Computer Science

Abstract

This project explores an approach to automatically creating systems from formal specifications. The ultimate goal is to allow programmers or system designers to write down a declarative specification of the properties they want their system to have and then to automatically synthesise a correct-by-construction system or controller which is guaranteed to satisfy these properties. In particular, this project uses an approach which models a system that can interact with its environment and has a kind of resource or energy that it can spend or gain. The synthesised system must then satisfy any properties in the specification while ensuring it never runs out of resources. This report introduces the background and basic theory, before designing and implementing new algorithms to solve a certain form of this synthesis problem.

Statement of Originality

I have acknowledged all sources, and identified any content taken from elsewhere.

I did all the work myself, or with my allocated group, and have not helped anyone else.

The material in the report is genuine, and I have included all my data/-code/designs.

I have not submitted any part of this work for another assessment.

My work did not involve human participants, their cells or data, or animals.

I have used the following open source code: the OCaml system, CUDD, MLCuddIDL. The use of this code is explained and referenced in the project.

Contents

Chapter 1

Introduction

The goal of reactive synthesis is to take a precise specification of a system and automatically produce a correct-by-construction finite state machine which satisfies this specification. Traditionally, these specifications are purely Boolean; either a system satisfies the required logical properties or it doesn't. Often, however, there are multiple different valid implementations of the same specification, and we would like to distinguish between "better" and "worse" implementations. One way to do this is to allow the specification to include a notion of "resources". The idea is that the system will either gain or consume resources depending on which actions it takes. Then, the synthesised implementation must satisfy the basic Boolean specification while also ensuring that it never runs out of resources. It is then possible to define a what an optimal implementation of such a specification means. In this case, it is an implementation which requires the smallest amount of starting resources in any given starting state.

Reactive synthesis is a difficult problem, as the state spaces involved quickly become intractable for even modestly sized specifications. Adding quantitative notions like resources makes this problem even worse. However, it turns out that, for specifications of a certain form, it is possible to represent the state space implicitly (symbolically) using binary decision diagrams (BDDs), saving the need to explicitly instantiate the entire state space.

This project presents a tool which can synthesise resource-aware systems entirely symbolically using BDDs. It involves: a language to express the required form of specification; a parser and compiler to convert these specifications into BDDs and a solver to produce the implementation.

The original contribution of this project consists in the design of new symbolic synthesis algorithms, based on a combination of those found in [Cîr19] and [MPR16]. The symbolic BDD based algorithms in [MPR16] do not deal with specifications of goal states while the algorithms in [Cîr19] are intended for explicit state representations only. So, the algorithms used in this project were adapted and extended from those two papers to achieve symbolic synthesis with goal states.

Chapter 2

Background and Literature Review

This section gives an overview of how reactive synthesis can be framed in terms of two player games. It also introduces energy games, a quantitative extension of these games as well as a compact representation of energy games using BDDs. Finally, a short review of other tools which aim to tackle the quantitative synthesis problem is given.

2.1 Two Player Games

A game graph [BCJ18] consists of a set of states Q along with a set of directed edges E between these states. In a two player game, there is a system player (player 0) and an environment player (player 1). Q is divided into system states Q_{sys} and environment states Q_{env}. A simple example of a game is shown in Figure 2.1. A play of the game starts by placing a pebble on one of the states. In each round of the game, if the pebble is on a system state, the system player chooses an outgoing edge along which to move the pebble. Likewise for environment states. A play then consists of the infinite path formed by a sequence of these moves. A strategy for the system player of a game is a function $\sigma : Q^*Q_{sys} \rightarrow Q$. That is, given a prefix of a play consisting of zero or more states (denoted by Q^*) ending in a system state, it provides a successor state. σ must always choose a move that corresponds to an edge, so for any partial play $p = q_0q_1...q_n$ with $q_n \in Q_{sys}$, $(q_n, \sigma(p)) \in E$. A strategy is **positional** if it only depends on the current state (i.e. it is a

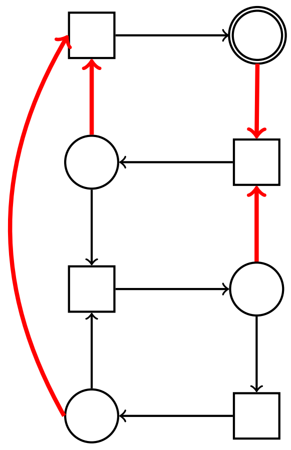


Figure 2.1: A simple two player game; environment states Q_{env} are denoted by squares, system states Q_{sys} by circles and the accept state Q_{win} by a double circle. The bold, red arrows show a winning system strategy.

function $\sigma : Q_{sys} \to Q$).

2.1.1 Büchi Games

It's possible to add different kinds of winning conditions to the two player games described above. A Büchi win condition defines a subset of states $Q_{win} \subseteq Q$. For a strategy of the system player to be a winning strategy, it must guarantee that any play it produces visits at least one state in Q_{win} infinitely often. The bold, red edges of Figure 2.1 correspond to a winning strategy of the Büchi game (where the only goal state is shown with a double circle).

2.2 Energy Games

Energy games [CD12] take the concept of two player games as defined above and add the concept of a resource, or energy. Weights are added to each edge, defined by a weight function $w : E \to \mathbb{Z}$. Then, a play starts as before, but with some initial energy. When an edge e is taken, $w(e)$ is added to the current energy of the play. Therefore, the energy is reduced if the edge has a "cost" (negative weight) and is increased if the edge has a positive weight. To win the game, the system must now ensure that the energy never goes

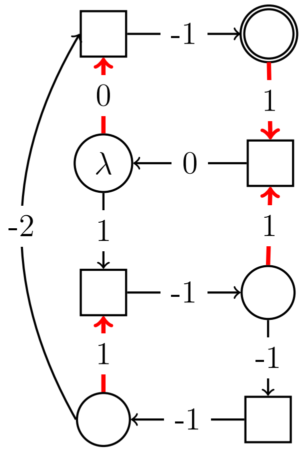


Figure 2.2: The game of Figure 2.1 extended with energies.

below zero throughout the play. Figure 2.2 shows one way of making the example in Figure 2.1 an energy game.

2.2.1 The Minimum Energy Problem and Strategies

The minimum energy problem asks what the smallest possible initial energy needed is at every node to allow the system player to have a winning strategy. The strategy shown in red in Figure 2.2 is the strategy for this game that requires the smallest starting energy for each system state. Note that it is different to the strategy for the unweighted game in Figure 2.1. In fact, the starting energy required is zero for every system state except the one labelled λ, which requires a starting energy of one.

2.3 Binary Decision Diagrams

Binary decision diagrams (or BDDs) are data structures which can represent boolean functions of multiple variables very compactly. An example of the BDD for the formula $x \wedge y$ is shown in Figure 2.3. To evaluate a particular assignment of the variables of a BDD, simply start at the root node (in this case x) and follow the solid line if the variable is assigned true and follow the dotted line if it is false. The terminal node you reach, either $\top$ for true or $\bot$ for false, tells you the truth value of the overall formula for that particular assignment. For example, if we let $x = \mathit{true}$, $y = \mathit{false}$, then we take the

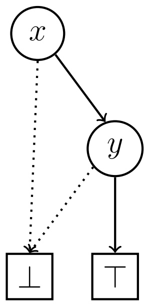


Figure 2.3: The BDD for $x \wedge y$

solid line from x and then the dotted line from y, finishing at $\perp$ as we expect (since $true \wedge false \equiv false$).

BDDs have several nice properties. Particularly; it's easy to perform operations on them, for example taking two BDDs f and g and computing $f \wedge g$ or computing $\neg f$ etc. Likewise, testing semantic equality of two BDDs amounts to simply comparing their structure. A BDD can be used to represent a set of states [MPR16]. Let $V_{var} := \{0,1\}^{var}$ represent all possible assignments to the variables var. Then, a state $s \in V_{var}$ is a particular assignment to all the variables in var. This state can be represented by a function $\chi_s : V_{var} \rightarrow \{0,1\}$ which evaluates to 1 (true) for s and 0 (false) for any other assignment. A set of states can then be represented in a similar way; by a function which evaluates to true for only the assignments in the desired set of states. This means that a set of states can be thought of as a function from boolean variables to either true or false and therefore can be represented by a BDD.

2.4 Symbolic Representation of Games and Strategies

It's possible to represent certain kinds of energy games (known as reactive energy games) using only boolean formulas, and therefore BDDs. In a reactive energy game, there is a set of boolean variables V divided into input (or environment) variables V_{env} and output (or system) variables V_{sys}. Each state of the game then consists of an assignment to all variables in V, so a state belongs to both the system and the environment (as opposed to the earlier energy games where a state belongs to one or the other). There are then

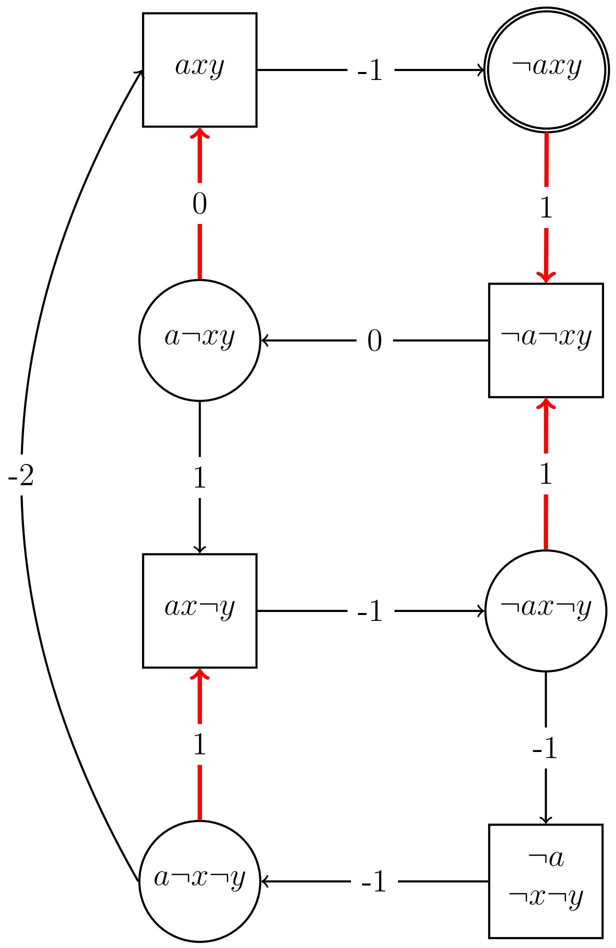


Figure 2.4: The game of Figure 2.2 with assignments to variables at each node.

two sets of transitions: ρ^e, the environment transitions and ρ^s, the system transitions. An environment transition is defined as a boolean formula over the set of variables $V_{env \cup sys \cup env'}$, where the primed variables $V_{env'}$ represent the environment variables in the next state (where the transition ends). System transitions are similarly boolean formulas, but are instead defined over $V_{env \cup sys \cup env' \cup sys'}$. The intended meaning is that, to make a move from a state with some assignment to $V_{env \cup sys}$, first the environment picks an assignment for $V_{env'}$ so that the resulting assignment to $V_{env \cup sys \cup env'}$ satisfies ρ^e. Then, the system picks an assignment for $V_{sys'}$ such that the overall assignment to $V_{env \cup sys \cup env' \cup sys'}$ satisfies ρ^s. That is, the environment picks a move first and the system responds, always taking strictly alternating turns.

It just so happens that the game in Figure 2.2 can be expressed as a reactive energy game. If we let $V_{env} = \{a\}$ and $V_{sys} = \{x, y\}$, then we can label each state with an assignment to these variables. Figure 2.4 shows such an assignment (where "$a \neg x y$" means $a = true, x = false, y = true$ etc.). Now, if we take the transition sequence starting at the top left state:

$$axy \xrightarrow{-1} \neg axy \xrightarrow{1} \neg a \neg xy$$

this can be expressed with the following transition relations:

$$\rho^e = (a \wedge x \wedge y) \rightarrow \neg a'$$
$$\rho^s = (a \wedge x \wedge y) \rightarrow (\neg a' \wedge \neg x' \wedge y')$$

The overall weight of this transition can also be represented as a pair of the weight and the formula. In this case, the overall weight is $(-1) + 1 = 0$, so the pair is:

$$(0, a \wedge x \wedge y \wedge \neg a' \wedge \neg x' \wedge y')$$

Since environment and system states strictly alternate in this game, it's possible to represent every transition sequence of the form $\square \rightarrow \bigcirc \rightarrow \square$ in this way, and therefore the entire game can be represented by such boolean formulas. This leaves the following representation of a reactive energy game using BDDs:

$$\rho^e : V_{env \cup sys \cup env'} \rightarrow \{0, 1\}$$
$$\rho^s : V_{env \cup sys \cup env' \cup sys'} \rightarrow \{0, 1\}$$
$$w \in (\mathbb{N} \times V_{env \cup sys \cup env' \cup sys'} \rightarrow \{0, 1\})$$

Reactive games also have a definition of the allowed starting states:

$$\theta^e : V_{env} \to \{0, 1\}$$
$$\theta^s : V_{env \cup sys} \to \{0, 1\}$$

For this project, I also add system goal states (or liveness requirements) to the definition:

$$Q_{goal} : V_{env \cup sys} \to \{0, 1\}$$

2.5 BDD algorithms

The symbolic algorithms in this project are largely based on the BDD algorithm in [MPR16]. That algorithm calculates the minimum required starting energy for each state in a reactive energy game represented by BDDs (as described above). However, they do not provide a method for extracting a system strategy that realises this miniumum energy. Also, the reactive energy games considered there do not have Büchi goal states, so essentially every state is a goal state. The algorithms in [Cîr19] address both of these. However, those algorithms work for explicit state representations of the game graphs, not BDD representations.

So, this project essentially extends the BDD algorithm of [MPR16], using the algorithms in [Cîr19] to perform strategy synthesis for an energy game with Büchi goals entirely symbolically (i.e. using BDDs).

The "Weighted Büchi Games with Offsetting" considered in [Cîr19] are slightly different to energy games. The weights on edges can only be costs (they always reduce energy) and states can have "offsets" which increase energy. For simplicity, I only use the terminology of energy games in this project, but I believe it would be possible to reframe the symbolic games and algorithms in terms of WBAOs.

2.6 Synthesising a System

The choice of using reactive energy games of the form described above is, of course, not arbitrary. If the environment variables V_{env} are thought of as inputs to a system and the system variables V_{sys} are thought of as the outputs, then the environment transitions ρ^e can be thought of as assumptions about the behaviour of the environment and the system transitions ρ^s can be thought of as safety guarantees the system must not violate. Also, the goal states Q_{win} can be thought of as liveness guarantees that the system must meet. Along with weights, this gives a specification for a system that interacts with its environment and has some kind of energy it must keep above zero as it chooses its moves. Therefore, a winning system strategy for a reactive energy game is actually an implementation of a system that meets all of the necessary guarantees and makes sure it never runs out of energy. A concrete example of a specification that vaguely approximates a useful system is shown in Section 3.1.

2.7 Existing Tools

2.7.1 PRISM

PRISM [KNP11] is a probabilistic model checker used mostly for verification of probabilistic systems. Some work has been done [Gia+18] to implement strategy synthesis in PRISM. In PRISM, rewards can be defined for taking specific actions. A strategy is then synthesised which looks to maximise the expected reward gained in a given probabilistic environment. This approach deals with maximising rewards rather than minimising required energy.

2.7.2 Slugs

Slugs [ER16] is a tool that implements Generalised Reactive(1) (GR(1)) synthesis [Blo+12], which uses very similar kinds of specifications to those used in this project, but it does not deal with quantitative properties by default. It includes some extensions, notably one which implements a two dimensional cost notion (detailed in [JEK13]). This allows transitions in a 2D space (eg. a robot moving in a warehouse) to be given transition costs. The GR(1) algorithm is then extended to prefer transitions which have a lower cost to reach the next goal state, hence making the strategy produced optimal in terms of transition cost to reach the goals. The extension also penalises strategies which excessively wait for environment conditions to be met before they can take a transition. Slugs does not, however, include any notion of a “resource” that can be spent and gained.

2.7.3 QUASY

QUASY [Cha+11] is the tool that takes the most similar approach to this project. It considers the following situation: a set of input variables and output variables are given along with a linear temporal logic (LTL) specification which must be satisfied by the resulting system for any sequence of inputs. This is the classic, simple version of reactive synthesis. QUASY then also requires a weighted “mean payoff” automaton (MPA) to be defined. This automaton has transitions over both input and output variables and these transitions have non-negative weights associated with them. It must also be deterministic and complete.

The point of this MPA is to judge the “quality” of an implementation which satisfies the specification. Given a particular sequence of inputs, an implementation will produce a corresponding sequence of outputs forming a sequence over both input and output variables. This sequence can then be used as an input for the MPA and the quality is defined as the mean of the weights of the transitions taken by the MPA over the length of the sequence (the “mean payoff”).

In [Blo+09], a method is given for synthesising an implementation which produces the best mean payoff in the worst possible case. This is done as follows: the LTL specification is represented as a deterministic parity automaton and the quantitative specification is given as an MPA; these two automata are combined into an automaton which accepts the same language as the parity automaton while giving the same mean-payoff for each word as the MPA. This resulting automaton is then converted into a two player game by splitting each of its transitions (which are over both the input and output variables) into two: one over the input variables with and one over the output variables with a new state in between. These new states are the system (player 1) states of the game while the original states are the environment states (player 2). Solving this mean payoff game results in a strategy that can be used to create the optimal synthesised system.

QUASY can either optimise the system to have the greatest mean-payoff in the worst possible case, or it can optimise for the most likely case (given by a probability distribution). These notions of optimality, which seek to maximise mean-payoff, are different from the notion used in [Cîr19], which seeks to minimise initial required resources.

Chapter 3

Design

3.1 Specification Language

The specification language was designed to be as simple as possible while still being expressive enough to easily write down useful specifications. To this end, the language is mostly influenced by the input language of Slugs (see Section 2.7) as well as the recommendations of [LP99]. Defining a specification more or less amounts to writing down the definition of a symbolic reactive energy game (described in Section 2.4). An example of a specification for a simple lift controller is shown in Figure 3.1. The environment has the ability to request a destination floor by setting `pending` to true and `dest_floor` to the desired floor. The system can control which way the lift moves in the next step by setting `move` to 0 for down, 1 for stop and 2 for up.

Lines 1-5 declare the environment variables. A variable is either boolean, like `pending`, in which case it is simply declared by itself, or it is a bounded integer like `src_floor` in which case it needs bounds to be defined as in Line 3. Then, the environment transitions (or assumptions) are defined. Line 11 ensures the source floor is set when a floor is requested to remember where the lift started. Line 12 ensures that the source floor and destination floor don't change while a request is unfulfilled (i.e. it makes sure they are remembered). Lines 13 and 14 make sure that pending stays true until the destination floor is reached, at which point it is set to false. Lines 15-17 move the current floor up or down depending on the move variable chosen by the system.

The system transitions on Lines 19-21 ensure the system doesn't try to move down on the bottom floor or up on the top floor. The only system goal (Line 24) says that the system must guarantee that the destination floor is always eventually visited.

The weight definitions (Lines 26-35) assign a weight of -1 to any transition while a request is pending and a positive weight equal to the distance between the starting floor and the destination floor when a request is fulfilled.

This specification doesn't have any initial states defined, but they would be under the headings of `[env_init]` and `[sys_init]` before the transitions are defined.

```
[env_vars]
pending
src_floor     : [0..4]
dest_floor    : [0..4]
current_floor : [0..4]

[sys_vars]
move          : [0..2]

[env_trans]
!pending & pending' -> src_floor' = current_floor'
pending -> (src_floor' = src_floor & dest_floor' = dest_floor)
pending & !(current_floor = dest_floor) -> pending'
(current_floor = dest_floor) & pending  -> !pending'
move = 2 & current_floor < 4   -> current_floor' = current_floor + 1
move = 1                       -> current_floor' = current_floor
move = 0 & current_floor > 0   -> current_floor' = current_floor - 1

[sys_trans]
current_floor = 4 -> !(move = 2)
current_floor = 0 -> !(move = 0)

[sys_goals]
pending & !pending'

[weights]
pending & !(current_floor = dest_floor) : -1
pending & !pending' & (src_floor - dest_floor) = 1
    | (dest_floor - src_floor) = 1 : 1
pending & !pending' & (src_floor - dest_floor) = 2
    | (dest_floor - src_floor) = 2 : 2
pending & !pending' & (src_floor - dest_floor) = 3
    | (dest_floor - src_floor) = 3 : 3
pending & !pending' & (src_floor - dest_floor) = 4
    | (dest_floor - src_floor) = 4 : 4
```

Figure 3.1: An example specification of a simple lift controller

3.2 Algorithms

3.2.1 Calculating Minimal Required Energies

As described in Section 2.5, my algorithm for calculating the minimal required starting energy for each state is a combination of the BDD algorithm given in [MPR16] (Algorithm 2) and the extent algorithm in [Cîr19] (Figure 1). The original BDD algorithm in [MPR16] does not include goal states (Büchi conditions or liveness guarantees) at all. On the other hand, the algorithms in [Cîr19] work with explicit state representations of games, not symbolic representations. So, much of the work of this project was in combining the two approaches to obtain a new symbolic BDD based algorithm that could take goal states into account. The result is shown in Algorithm 1.

The algorithm uses nested fixed points to calculate minimum energies for each state (explained fully in [Cîr19]). It is first called with the parameters $accept = true, engs = \emptyset$. *accept* is set to true if we are calculating (a least fixpoint of) the minimum energies for the accept states and false if we are calculating (a greatest fixpoint of) the minimum energies for the non-accept states. Accordingly, in Line 2, C is set to the states for which we want to calculate minimum energies. In Line 3, the minimum energies for all states in C are set to 0 if they are the accept states and ∞ if they are the non-accept states. Essentially, this means that we start off by assuming that, if we are in an accept state, we can reach an accept state with zero energy. Likewise, if we are not in an accept state, we start by assuming we cannot reach an accept state. The fixpoint is then evaluated in the loop from Lines 4-21, which continues until the minimum energies for C no longer change.

First, the minimum energies from the previous iteration are saved in Line 5. Then, in Lines 6-8, if *accept* is true, we calculate a best approximation for the minimum energies of non-accept states given the current minimum energies we have calculated for the accept states. Line 9 removes any mapping for the minimum energies of C, so that they can be recalculated. Set in Line 10, *remaining* is the set of states we have not yet found a new extent value for in this iteration. The loop from Lines 11-20 iterates over increasing putative extent values *best*. Within this loop, in Lines 13-16 we iterate over every possible weight associated with a transition. We then calculate the set of states S. These are the states which have a current estimated extent e_v for which there exists some transition T with weight v such that $e_v - v \leq best$. This means that there could be some transition which would end in S and start in a state with a minimum initial energy less than best. $bestT$ is the subset of transitions in T which end in S. We then, in Line 17, want to find the states B for which the system can force the environment to take some transition in $bestT$. This is calculated with **forceEnvTo** which is implemented in BDD operations as $(\rho^e \implies (\rho^s \wedge bestT)_{\exists sys'})_{\forall env'}$ as in [MPR16]. This formula finds the states (in $V_{env \cup sys}$) where, for every possible environment move ($\forall env'$), if this move is allowed by ρ^e then there must exist some system move ($\exists sys'$) which is allowed by ρ^s so that the overall transition is in $bestT$. The intersection with *remaining* is also taken so that we only consider states for which we have not already set a minimum energy. We then add a mapping from *best* to B to the minimum energy mapping and remove B from remaining so that we don't set an extent for B again.

In this algorithm, Lines 4-21 are almost identical to the BDD algorithm of [MPR16]. The major changes I made are Lines 2, 3 and 7 which allow calculating the nested fixpoints found in [Cîr19], therefore accounting for accept and non-accept states. The major difference between this algorithm and the one in [Cîr19] is the fact that, in that algorithm, every recursive call iterates over every state while in this algorithm every recursive call iterates over possible energies (up to maxEng). In [Cîr19], the idea is that, for every state, we find the transition which, when taken, will require the smallest starting energy. So, if we have a state q, we want to find the transition with weight w to another state q' with current initial energy e' such that $e' - w$ is minimised (using the convention that negative weights are costs). For this algorithm, however, we iterate over increasing minimum energies b, and we want to find the set of states B which have some transition to another state such that $e' - w \leq b$. Therefore, the tradeoff for this algorithm is that we no longer need to iterate over the whole state space, but we do need to iterate over multiple energies (up to $maxEng$). In the worst case, every state has a different energy, so we do effectively need to iterate over the whole state space.

Algorithm 1 Computing the minimum required starting energy for each set of states.

Input: A symbolic game with environment transitions ρ^e, system transitions ρ^s, $\mathit{weights} \in (\mathbb{Z} \times \text{Set of Transitions})$, goal states Q_{goal}, and an energy bound $\mathit{maxEng} \in \mathbb{N}$

Output: Minimum energies required $\in (\{\mathbb{N} \cup \infty\} \times \text{Set of States})$

```
1:  procedure MinEng($accept \in Bool, engs \in (\{\mathbb{N} \cup \infty\} \times States)$)
2:      $C \leftarrow \begin{cases} Q_{goal} & \text{if } accept = true \\ \neg Q_{goal} & \text{otherwise} \end{cases}$
3:      set $engs$ for $C$ to $\begin{cases} 0 & \text{if } accept = true \\ \infty & \text{otherwise} \end{cases}$
4:      repeat
5:          $old \leftarrow engs$
6:          if $accept = true$ then
7:              $engs \leftarrow$ MinEng($false, engs$)
8:          end if
9:          clear $engs$ for $C$
10:         $remaining \leftarrow \{s \in S \mid (e, S) \in old\} \cap C$
11:         for $best \in \{0..maxEng\} \cup \{\infty\}$ do
12:             $bestT \leftarrow \emptyset$
13:             for $(v, T) \in weights$ do
14:                 $S \leftarrow \{s \in S_{e_v} \mid (e_v, S_{e_v}) \in old \wedge e_v - v \leq best\}$
15:                 $bestT \leftarrow bestT \vee (T \wedge prime(S))$
16:             end for
17:             $B \leftarrow$ (forceEnvTo $bestT$) $\cap\ remaining$
18:             add $(best, B)$ to $engs$
19:             remove $B$ from $remaining$
20:         end for
21:     until $engs = old$
22:     return $engs$
23: end procedure
24:
25: MinEng($true, \emptyset$)
```

3.2.2 Extracting Strategies

While the algorithms in [MPR16] calculate minimum energies, they do not deal with extracting the strategies that actually realise those minimum required energies. So, my implementation takes roughly the same approach to extracting strategies as [Cîr19], adapted to work with BDDs. In general, a strategy could require a number of different moves for each state, with each move depending on the current energy level when a play of the game is in that state. In turns out, however (shown in [Cîr19]), that only one move is needed from a goal state, and only two moves are needed from non-goal states: one to increase energy (the good-for-energy strategy) and one to move towards a goal state (the attractor strategy). Then, for each state we obtain an energy threshold. During a run of the strategy, if the system has energy greater than or equal to the threshold, it uses the attractor strategy, otherwise it uses the good-for-energy strategy.

This strategy extraction is implemented in Algorithm 2. This algorithm starts by calculating the minimum required energies using Algorithm 1. Then, in Line 3, the strategy which gives the unique move out of each accept state is calculated using Algorithm 3. The attractor and good-for-energy strategies are then calculated as follows.

Intuitively, this part of the algorithm works by rerunning the last iteration of Algorithm 1 for non-accept states. So, the set of states B and transitions $bestT$ are as described in Section 3.2.1. From Line 4 onwards, this algorithm looks very similar to a call of Algorithm 1 with $accept = false$. That is, it sets the minimum energies for the non-accept states to ∞ and then recalculates the last fixpoint. Then, in Lines 19-20, for each set of states B (as in Algorithm 1), the set of states $attrB$ which have no attractor strategy set for them is found. The attractor strategy σ_a is then set to $bestT$ for these states and the threshold θ is set to the value of $best$ for these states. The attractor strategy must also adhere to the system and environment transitions, so a conjunction with these is taken. For the states $engB$ in B which already have an attractor strategy set, the good-for-energy strategy is set to $bestT$ (in Line 23), making sure to adhere to ρ^e and ρ^s as for the attractor. Finally, in Line 24, the states $noAttr$ for which no attractor strategy has been set are updated to no longer include $attrB$, as the attractor has now been set for these states.

Algorithm 2 Computing the attractor and good-for-energy strategies.

Input: A symbolic game with environment transitions ρ^e, system transitions ρ^s, $weights \in (\mathbb{Z} \times \text{Set of Transitions})$ and an energy bound $maxEng \in \mathbb{N}$

Output: Strategies $\sigma,\ \sigma_a,\ \sigma_e$: Transitions, threshold: $\theta \in (\mathbb{N} \times \text{Set of States})$

```
1: procedure STRATEGIES
2:     $engs \leftarrow \textsc{MinEng}(true, \emptyset)$
3:     $\sigma \leftarrow \textsc{AcceptStrategy}(engs)$
4:     set $engs$ for $\neg Q_{goal}$ to $\infty$
5:     $noAttr \leftarrow \neg Q_{goal}$
6:     repeat
7:         $old \leftarrow engs$
8:         clear $engs$ for $\neg Q_{goal}$
9:         $remaining \leftarrow \{s \in S \mid (e, S) \in old\} \cap C$
10:        for $best \in \{0..maxEng\} \cup \{\infty\}$ do
11:            $bestT \leftarrow \emptyset$
12:            for $(v, T) \in weights$ do
13:                $S \leftarrow \{s \in S_{e_v} \mid (e_v, S_{e_v}) \in old \wedge e_v - v \leq best\}$
14:                $bestT \leftarrow bestT \vee (T \wedge prime(S))$
15:            end for
16:            $B \leftarrow (\textbf{forceEnvTo}\ bestT) \cap remaining$
17:            add $(best, B)$ to $engs$
18:            remove $B$ from $remaining$
19:            $attrB \leftarrow B \cap noAttr$
20:            $\sigma_a \leftarrow \sigma_a \vee (attrB \wedge bestT \wedge \rho^e \wedge \rho^s)$
21:            add $(best, attrB)$ to $\theta$
22:            $engB \leftarrow B \wedge \neg noAttr$
23:            $\sigma_e \leftarrow (\sigma_e \wedge \neg engB) \vee (engB \wedge bestT \wedge \rho^e \wedge \rho^s)$
24:            $noAttr \leftarrow noAttr \wedge \neg attrB$
25:        end for
26:    until $engs = old$
27:    return $\sigma,\ \sigma_a,\ \sigma_e,\ \theta$
28: end procedure
```

Algorithm 3 Computing the accept state strategies.

```
1:  procedure AcceptStrategy
2:      $remaining \leftarrow Q_{goal}$
3:      for $(e, S) \in engs$ do
4:          $O \leftarrow S \cap remaining$
5:          for $(v, T) \in weights$ do
6:              $P \leftarrow \{r \in R | (n, R) \in engs \wedge (n = max(0, e + v))\}$
7:          end for
8:          $thisT \leftarrow T \wedge prime(P)$
9:          $B \leftarrow (\textbf{forceEnvTo } thisT) \cup O$
10:         $\sigma_a \leftarrow (\sigma_a \wedge \neg B) \vee (B \wedge thisT \wedge \rho^e \wedge \rho^s)$
11:         $O \leftarrow O \wedge \neg B$
12:         $remaining \leftarrow remaining \wedge \neg B$
13:     end for
14: end procedure
```

Algorithm 3 computes the moves the system should take from accept states. This essentially consists of finding a transition with weight v from each accept state s with minimum energy e to another state r with minimum energy n such that $n = max(0, e + v)$ (Line 6). The strategy is then computed in a similar manner to the attractor and good-for-energy strategies in Algorithm 2.

Chapter 4

Implementation

4.1 Tools and Libraries

This project was implemented in OCaml[1], a statically typed functional language in the ML family. I chose it mainly because of its functional style and expressive type system which allows very easy representation and manipulation of ASTs, as well as the ability to ensure certain properties (especially of AST transformations). It also has excellent integration with Emacs, used as a development environment.

The lexing and parsing were done with ocamllex and Menhir, a lexer and parser generator respectively for OCaml.

The creation and manipulation of BDDs was done using MLCuddIDL[2] which provides an OCaml interface to the C library CUDD, a popular and mature BDD library used in many verification applications.

Git and GitHub were used for version control, providing all of the usual benefits of a good VCS.

4.2 Overall Structure

The program is mostly contained in the following modules:

- **lexer.mll**: The code to generate the lexer.

[1]https://ocaml.org/

[2]https://www.inrialpes.fr/pop-art/people/bjeannet/mlxxxidl-forge/mlcuddidl/index.html

```
type formula =
  | And of (formula * formula)
  | Or of (formula * formula)
  | Not of formula
  | Atom of atom

and atom =
  | Lt of (term * term)
  | Gt of (term * term)
  | Eq of (term * term)
  | Bvar of string
  | Bool of bool

and term =
  | Add of (term * term)
  | Sub of (term * term)
  | Ivar of string
  | Int of int
```

Figure 4.1: Abstract syntax tree for specification expressions

- **parser.mly**: The code to generate the parser.
- **syntax.ml**: The abstract syntax tree used to represent a specification.
- **game.ml**: The functions to convert the AST into a symbolic game, consisting of a record type with the required BDDs.
- **strategy.ml**: The implementations of the central algorithms to calculate the minimum energies of and to solve the symbolic game.

4.3 Compiling to BDDs

Thanks to OCaml's excellent type system, the AST of the specification language can be represented directly with algebraic data types. These types are shown in Figure 4.1.

Simple boolean formulas are straightforward to convert into BDDs. However, bounded integer operations are more difficult. So, it is necessary to compile the raw AST from expressions involving arithmetic operations to purely boolean formulas which can be represented with BDDs.

For this, I used the approach described in Chapter 6 of [KS08]. Briefly, it involves encoding integers and integer variables as bit vectors. Then, integer addition can be implemented by encoding the circuit that would add two bit vectors i.e. a chain of full adders, also known as a ripple carry adder. Subtraction and relational operators ($<, =, \neq$, etc.) can be obtained in a similar way.

For bounded integer variables, their bounds must be enforced as constraints. So, for an environment variable $v : [x..y]$, the constraint $(v' >= x) \wedge (v' <= y)$ is added to the environment transitions. Also, the constraint $(v >= x) \wedge (v <= y)$ is added to the initial environment states, to ensure the game can't start in an illegal state. Likewise for a bounded system variable.

The weight definitions in the specification also require some preprocessing. This is because two different weight definitions can overlap. In this case, their intersection is simply assigned the sum of their weights. Any transition which doesn't have a weight defined is given a weight of zero.

4.4 Solving the Game

The resulting game is represented as a record type with the following form (omitting some implementation details):

```
type t = {
    env_init : bdd;          (* Initial environment states *)
    sys_init : bdd;          (* Initial system states *)
    env_trans : bdd;         (* Environment transitions *)
    sys_trans : bdd;         (* System transitions *)
    sys_goals : bdd;         (* System goal states *)
    weights : bdd Map.M(Int).t  (* Weight definitions *)
}
```

Figure 4.2: Data type for a symbolic game

Here, `bdd Map.M(Int).t` is a map from integers (weights) to BDDs (transitions). This, along with a prime function that takes a BDD and maps all of its variables to their primed equivalents, is all that is needed for a fairly straightforward implementation of Algorithms 1 and 2. The result is a strategy with the type:

```
type t = {
    acceptStrategy : Game.bdd;
    attractor : Game.bdd;
    goodForEnergy : Game.bdd;
    threshold : Game.bdd Map.M(Int).t;
    minEnergy : Game.bdd Map.M(TropicalNat).t;
}
```

Figure 4.3: Data type for a symbolic strategy

`TropicalNat` is a type that consists of $\{\mathbb{N} \cup \infty\}$. This is because it's possible for some states to have no winning strategy when the game starts from them, therefore they have no minimum initial energy and are assigned to ∞.

Chapter 5

Planning and Testing

5.1 Fulfilment of Original Plans

A comparison of the original Gantt chart from my progress report and the actual schedule is shown in Appendix A. Clearly, plans were somewhat altered by the Great Plague, but I still made substantial changes from the original brief (see Appendix D) and progress report. Most notably, the plans in both of these had the intention of only implementing the explicit state algorithms in [Cîr19] with little or no modifications. This would have included an interactive mode for the user to take the role of the environment and run the resulting strategy. However, when starting to implement these explicit state algorithms, I realised that fully symbolic algorithms would likely be possible. These symbolic algorithms are what I then designed and implemented. This did require a significant time investment, so I was not able to include the interactive mode described in the progress report. I do still believe that such a mode would be very useful both for understanding the resulting strategies and for testing them (potentially in an automated way with random inputs). So, this mode is consigned to future work.

I also decided to implement this tool using OCaml rather than Java, mostly as it provides better support for BDD manipulation, as well as for reasons described in Section 4.1.

5.2 Testing

5.2.1 A Simple Example

As a first informal validation of the algorithm and implementation, I adapted a very simple specification (taken from [ER16]) to check that the correct strategy was produced, shown in Figure 5.1. This specification says that a' must be set to true by the environment unless x and y are both false. The system can only set one of x' or y' to be true and the system must reach a state where $a \wedge y$ is true infinitely often. The resulting strategy for accept states is represented as a BDD with the formula:

$$!y' \wedge a' \wedge y \wedge a$$

This is set of allowed assignments to current variables and primed variables that the system must satisfy when moving from an accept state. For example, $!y'$ means that y' must be false in the chosen transition. As expected, $y \wedge a$ must be true, as this is the definition of being in an accept state. a' must be true, since this is required by the environment transition. y' must then be set to false by the system, as having $y = true$ will cost -1 energy in the next transition, so this would not be the lowest energy transition out of the accept state. Any variables not mentioned (i.e. b and x) are allowed to take any value. The full strategy with the attractor, good-for-energy, thresholds and minimum energies is given in Appendix C. I manually verified this resulting strategy to ensure that it is correct.

```
[env_vars]
a
b

[sys_vars]
x
y

[env_trans]
a' | !(x | y)

[sys_trans]
!x' | !y'

[sys_goals]
a & y

[weights]
y     :  -1
x     :  1
```

Figure 5.1: A simple example specification

5.2.2 Expect Testing

Testing was carried with Cram-style[1] testing using Expect-test[2]. Expect-test is a framework that allows writing inline tests along with source code in the same files. These tests are intended to print an output which is then compared against an expected output. A simple example test is shown in Figure 5.2. This approach to testing makes writing unit tests very simple, as all that's needed is to define how to print the output of interest. It also provides regression testing very cheaply, as these inline tests can be run whenever changes are made.

[1]https://bitheap.org/cram/

[2]https://github.com/janestreet/ppx_expect

```
let%expect_test "parse_sys_trans_only" =
  let input = {|
    [sys_trans]
    !x & !y
  |}
  in
  let lexbuf = Lexing.from_string input in
  let result =
    match parse_with_error lexbuf with
    | Some tree -> List.map ~f:Syntax.show_formula tree.sys_trans
                   |> String.concat;
    | None -> "No parse tree produced";
  in
  printf "%s" result;
  [%expect {|
    (Syntax.And
       ((Syntax.Not (Syntax.Atom (Syntax.Bvar "x"))),
        (Syntax.Not (Syntax.Atom (Syntax.Bvar "y"))))) |}]
```

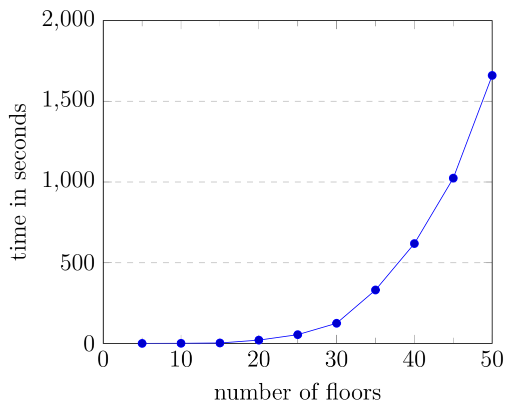
Figure 5.2: An example expect-test, checking the correct parsing of a specification

Chapter 6

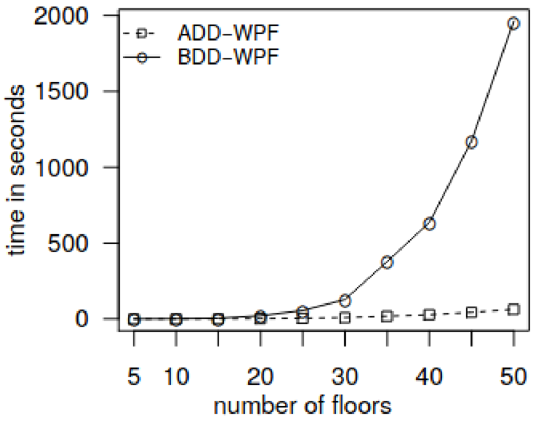
Evaluation

When there are no system goals defined, this project's implementation should be more or less identical to [MPR16]. So, I recreated the initial evaluation in that paper as a way to validate this one. Specifically, I used their "WPF" lift specification which is the same as the example given here in Figure 3.1 minus the system goals. Their evaluation tests the scaling of their algorithms as the number of states (number of lift floors) increases. This is simply done by taking the basic lift specification, increasing the top floor, and adding a weight definition for each floor (as in Figure 3.1). All evaluations for this project were using CUDD 3.0.0 and run on a 3.5GHz Intel i5 CPU, 8GB RAM on NixOS 20.03. The results are shown side by side with the original in Figure 6.1. Each data point was run 5 times and an average taken. CUDD automatic variable reordering was also turned off to reduce variance in the run times. Perhaps somewhat surprisingly, the results for this implementation are almost identical to the original paper. My implementation also finds the same worst case initial required energies for a lift with five floors as is mentioned in that paper. So, these results, along with a number of tests (discussed in Section 5.2) gives high confidence that the algorithms and implementation in this project are correct for the case of no system goal states.

For completeness, Figure 6.2 shows the running times for a lift specification with only two weights defined (see Figure B.1 and Listing 3 in [MPR16]). As can be seen, the number of different weights defined makes a significant difference to the running time of the algorithm, with a specification with only two weights being an order of magnitude faster. This is because the time complexity of the BDD algorithm with no goal states is $O(N \cdot maxEng^2 \cdot |weights|)$ [MPR16] where N is the number of states.



(a) The results for this implementation



(b) Results from [MPR16]

Figure 6.1: A comparison of the running times of this project's implementation vs. that of [MPR16]

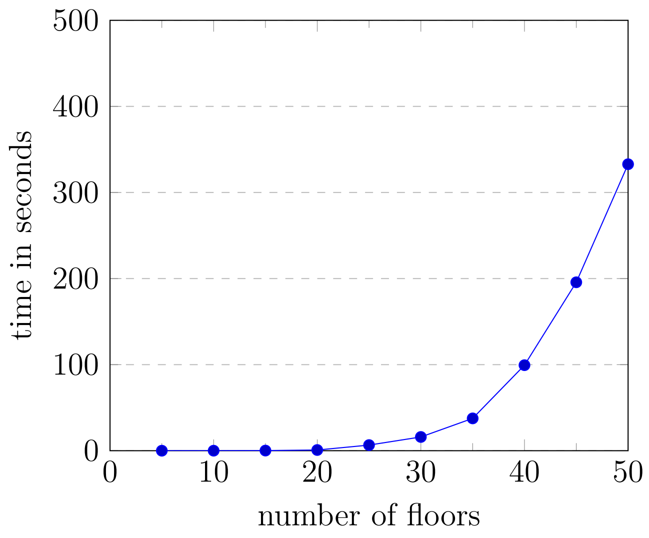


Figure 6.2: Running times of lift specifications with only two weights defined

6.1 Adding Goal States

To test the effect of adding goal states, I used a somewhat simpler lift specification (see Figure B.2). This specification is intended to represent a lift which can only “recharge” when on the bottom floor. Any other move costs one energy. Given this energy model, the system goal now requires that, when a floor is requested, it is eventually visited. The number of lift floors is scaled in a similar way to the initial evaluation above. The results are shown in Figure 6.3. The running times here are significantly worse than for a lift without goal states, even though only two weights are defined. I believe that the worst case time complexity is proportional to $maxEng^4$. This is because the accept state fixpoint can run for $O(N_{accept} \cdot maxEng^2 \cdot |weights|)$ steps, as above (except $N_{accept} = |Q_{accept}|$). Then each non-accept fixpoint can run for $O(N_{non-accept} \cdot maxEng^2 \cdot |weights|)$.

The high running time seen in Figure 6.3 seems to stem from the fact that there are some states for which there exists no winning strategy. However, all of these states are not legal states as they include values for bounded integers outside of the allowed bounds. This is due to details of the implementation which do not affect the resulting system strategy, but strongly affect the running time because they force every possible energy up to $maxEng$ to be tried. These states could likely be completely excluded with an improved implementation.

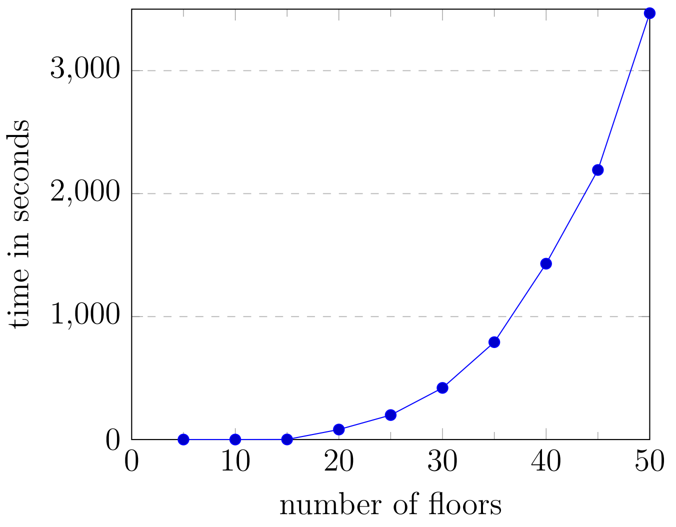


Figure 6.3: Running times of lift specifications with goal states

Chapter 7

Conclusions and Future Work

This project went reasonably far beyond the original plans, as it included the design of some original algorithms as well as a more complicated implementation. Some cosmetic features were not included (notably the intended interactive mode), but the essential functionality is complete and, to my knowledge, unique among other tools in its scope.

As well as the BDD algorithm extended in this project, [MPR16] also contains an algorithm which represents reactive energy games with a mixture of algebraic decision diagrams (ADDs) instead of only BDDs. An obvious extension of this work would be to develop similar algorithms which use this ADD based representation. Looking the results in that paper, this would greatly improve the worst case time complexity (although at the possible expense of using much more memory). It would then presumably also be possible to represent the resulting threshold function with an ADD, which might speed up strategy execution significantly.

It may well be possible to extend this approach to the strategy synthesis problem to other types of quantitative game. For example; mean payoff games or multi-dimensional energy games (a generalisation of energy games).

The evaluation included in this project is rather limited. It only deals with small, contrived examples that don't give much insight into the utility of this tool for any kind of real world usage. A major reason for this is that the language is restricted in its expressiveness, so even modestly complex specifications quickly become tedious and impractical to write out by hand. Future work would therefore include improvements to the language to allow a more natural style of expressing complicated specifications.

Along a similar vein, another direction to explore would be to actually integrate this tool with a real practical example such as a simple robot controller. Some similar work has been done (eg. [MR15]), but it would be very interesting to test the usefulness of adding energy constraints to such specifications.

Bibliography

[LP99] Leslie Lamport and Lawrence C Paulson. “Should your specification language be typed”. In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 21.3 (1999), pp. 502–526.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. 1st ed. Springer Publishing Company, Incorporated, 2008. ISBN: 3540741046.

[Blo+09] Roderick Bloem et al. *Better Quality in Synthesis through Quantitative Objectives*. 2009. arXiv: `0904.2638 [cs.LO]`.

[Cha+11] Krishnendu Chatterjee et al. “Quasy: Quantitative synthesis tool”. In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2011, pp. 267–271.

[KNP11] Marta Kwiatkowska, Gethin Norman and David Parker. “PRISM 4.0: Verification of probabilistic real-time systems”. In: *International conference on computer aided verification*. Springer. 2011, pp. 585–591.

[Blo+12] Roderick Bloem et al. “Synthesis of reactive (1) designs”. In: *Journal of Computer and System Sciences* 78.3 (2012), pp. 911–938.

[CD12] Krishnendu Chatterjee and Laurent Doyen. “Energy parity games”. In: *Theoretical Computer Science* 458 (2012), pp. 49–60.

[JEK13] G. Jing, R. Ehlers and H. Kress-Gazit. “Shortcut through an evil door: Optimality of correct-by-construction controllers in adversarial environments”. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Nov. 2013, pp. 4796–4802. DOI: `10.1109/IROS.2013.6697048`.

[MR15] Shahar Maoz and Jan Oliver Ringert. "Synthesizing a Lego Forklift Controller in GR(1): A Case Study". In: *SYNT*. 2015.

[ER16] Rüdiger Ehlers and Vasumathi Raman. "Slugs: Extensible GR(1) synthesis". In: *International Conference on Computer Aided Verification*. Springer. 2016, pp. 333–339.

[MPR16] Shahar Maoz, Or Pistiner and Jan Oliver Ringert. "Symbolic BDD and ADD Algorithms for Energy Games". In: *SYNT@CAV*. 2016.

[BCJ18] Roderick Bloem, Krishnendu Chatterjee and Barbara Jobstmann. "Graph games and reactive synthesis". In: *Handbook of Model Checking*. Springer, 2018, pp. 921–962.

[Gia+18] Ruben Giaquinta et al. "Strategy synthesis for autonomous agents using PRISM". In: *NASA Formal Methods Symposium*. Springer. 2018, pp. 220–236.

[KMS18] Jan Kretínský, Tobias Meggendorfer and Salomon Sickert. "Owl: A Library for ω-Words, Automata, and LTL". In: *Automated Technology for Verification and Analysis - 16th International Symposium, ATVA 2018, Los Angeles, CA, USA, October 7-10, 2018, Proceedings*. 2018, pp. 543–550.

[Cîr19] Corina Cîrstea. "Resource-Aware Automata and Games for Optimal Synthesis". In: *Proceedings Tenth International Symposium on Games, Automata, Logics, and Formal Verification, GandALF 2019, Bordeaux, France, 2-3rd September 2019*. Ed. by Jérôme Leroux and Jean-François Raskin. Vol. 305. EPTCS. 2019, pp. 50–65. DOI: 10.4204/EPTCS.305.4. URL: https://doi.org/10.4204/EPTCS.305.4.

Appendix A

Planning

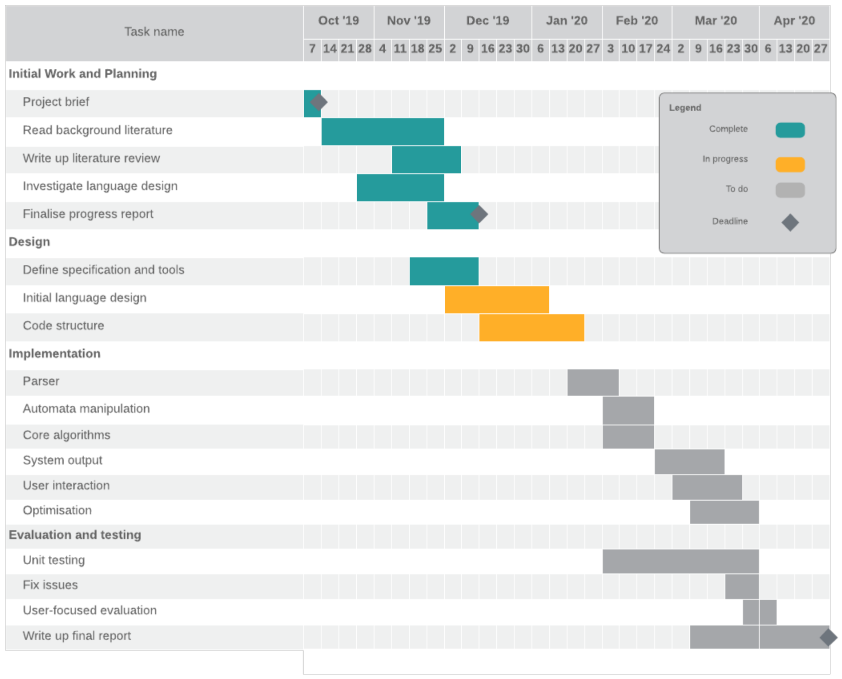


Figure A.1: Original Gantt chart from the progress report

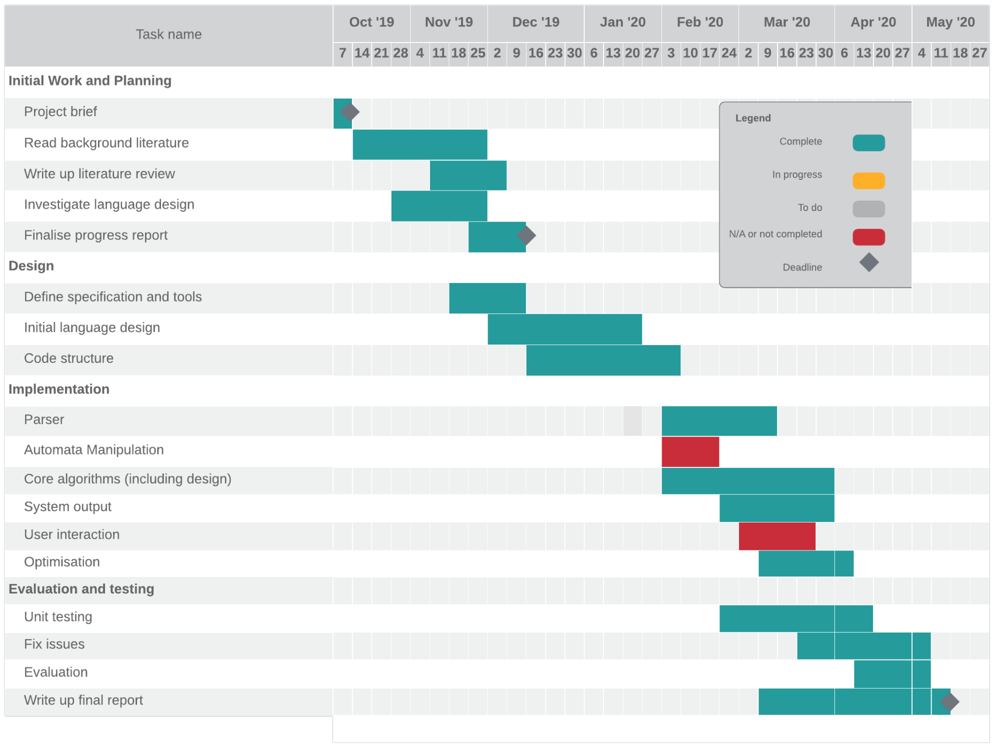


Figure A.2: Actual project progress as a Gantt chart

Appendix B

Evaluation

```
[env_vars]
pending
src_floor     : [0..$n]
dest_floor    : [0..$n]
current_floor : [0..$n]

[sys_vars]
move          : [0..2]

[env_trans]
!pending & pending'
-> src_floor' = current_floor'
pending
-> (src_floor' = src_floor & dest_floor' = dest_floor)
pending & !(current_floor = dest_floor) -> pending'
(current_floor = dest_floor) & pending  -> !pending'
move = 2 & current_floor < $n   -> current_floor' = current_floor + 1
move = 1
-> current_floor' = current_floor
move = 0 & current_floor > 0
-> current_floor' = current_floor - 1

[sys_trans]
current_floor = $n  -> !(move = 2)
current_floor = 0   -> !(move = 0)

[weights]
pending & !(current_floor = dest_floor) : -1
pending & (current_floor = dest_floor) : $n - 2
```

Figure B.1: Specification based on WTWO in [MPR16] for n + 1 floors

```
[env_vars]
pending
src_floor     : [0..$n]
dest_floor    : [0..$n]
current_floor : [0..$n]

[sys_vars]
move          : [0..2]

[env_trans]
!pending & pending'
-> src_floor' = current_floor'
pending
-> (src_floor' = src_floor & dest_floor' = dest_floor)
pending & !(current_floor = dest_floor) -> pending'
(current_floor = dest_floor) & pending  -> !pending'
move = 2 & current_floor < $n   -> current_floor' = current_floor + 1
move = 1
-> current_floor' = current_floor
move = 0 & current_floor > 0
-> current_floor' = current_floor - 1

[sys_trans]
current_floor = $n  -> !(move = 2)
current_floor = 0   -> !(move = 0)

[sys_goals]
!pending -> current_floor = dest_floor

[weights]
!(current_floor = 0) : -1
current_floor = 0 : 5
```

Figure B.2: Specification used to test the effect of adding goal states (for $n + 1$ floors)

Appendix C

Testing

C.1 A Simple Strategy

Figure C.1 shows the full resulting strategy for the specification in Figure 5.1. Unfortunately, the output format is not very pretty, but each expression defines the set of assignments which are allowed for that particular strategy (or set of states). $!a$ means the variable a must be false, $a \wedge b$ means both a and b must be true and $ITE(a; b; c)$ means: if a is true then b must also be true, else c must be true. $acceptStrategy$, $attractor$ and $goodForEnergy$ are strategies, so they define sets of allowed assignments over $V_{env \cup sys \cup env' \cup sys'}$. $threshold$ and $minimumEnergy$ define sets of states over $V_{env \cup sys}$ for each given threshold or minimum energy value.

```
acceptStrategy: !y'^a'^y^a

attractor:
ITE(a;
     !y^ITE(x;y'^!x'^a';!y'^x');
     ITE(x;
          y'^!x'^a';
          ITE(y;
               y'^!x'^a';
               !y'^x')))

goodForEnergy:
ITE(a;
     !y^ITE(x;
          a'^ITE(x';!y';true);
          !y');
     ITE(x;
          a'^ITE(y;
               !y';
               ITE(x';!y';true));
          !y'^ITE(y;a';true)))

threshold:
0: !y
1: y^x^!a
2: y^!x^!a

minimumEnergy:
0: ITE(x;true;!y)
1: y^!x
```

Figure C.1: A simple strategy

C.2 Test Cases

Module	Test	Passed?
parser.ml	Parsing the SLUGS example spec.	✓
	Parsing the SLUGS example spec. with weights	✓
	” ” a spec. with no trans.	✓
	” ” a spec with no goals	✓
	” ” a spec with simple bounded ints	✓
	” ” a spec with complex bounded ints	✓
	” ” the WPF lift spec	✓
	” ” the lift spec with goals	✓
game.ml	Creating a game from the SLUGS example spec.	✓
	” ” the SLUGS example spec. with weights	✓
	” ” a spec. with no trans.	✓
	” ” a spec with no goals	✓
	” ” a spec with simple bounded ints	✓
	” ” a spec with complex bounded ints	✓
	” ” the WPF lift spec	✓
	” ” the lift spec with goals	✓
strategy.ml	Solving the SLUGS example spec.	✓
	SLUGS example spec. has zero minEng	✓
	Solving the SLUGS example spec. with weights	✓
	SLUGS example spec. correct threshold and minEng	✓
	Solving a spec. with no trans.	✓
	Game with no goals has no attr./gfe	✓
	Game with no goals has zero minEng/thresh.	✓
	Solving a spec with simple bounded ints	✓
	Solving a spec with complex bounded ints	✓
	Lift spec. has correct extents	✓

Table C.1: The expect-test cases carried out

Appendix D

Original Brief

Problem

Reactive synthesis is a method of automatically generating an implementation of a given specification, satisfying all required properties, where the system must interact with its environment. Specifications are given in a temporal logic (usually LTL or similar) which are then translated into some form of automaton. Some tools exist which implement this kind of synthesis. Slugs[ER16] is an extensible framework for Generalised Reactive(1) synthesis with some support for computing cost-optimal implementations. QUASY[Cha+11] is a quantitative synthesis tool which can produce implementations for either worst-case or average-case environment behaviours, optimizing a quantitative specification. While these tools do aim to carry out optimal quantitative synthesis, this project would aim to implement a different notion of optimality, given in [Cîr19], where the initial resources required are minimised and both resource usage and resource gain are modelled.

Goals and Scope

The aim of this project is to create a simple prototype tool to perform optimal synthesis for resource-aware systems using the above notion of optimality. A simple language will be designed to describe the design space; system and environment variables as well as basic properties of states. It will include some basic types (i.e. Boolean, bounded integer etc.) as well as some operations on those types. As this tool will be a proof of concept, optimisation of the tool itself will not be a focus. As the desired properties will be represented as ω-automata, an existing tool such as OWL[KMS18] will be used to generate the automata from LTL formulae.