Aalborg University

Distributed and Embedded Systems

Semantics and Verification

EgGS - The Energy Game Strategizer

Master Thesis

Authors:
Mads V. Carlsen
Rasmus S. Jacobsen

Supervisors:
Kim G. Larsen
Erik R. Wognsen

10th June, 2014

AALBORG UNIVERSITY
STUDENT REPORT

Abstract

In this report we continue the work done in our previous semester report, on the subject of Energy Games. Energy games are games played in multiweighted automata with the goal of synthesizing a strategy for a controller, to keep the game running indefinitely. In doing so we present the implementation of a software tool designed to synthesize these strategies, named EgGS. Energy strategies can be used to control energy critical systems, such as satellites, robotics, etc.

In our previous report, a prototype implementation of this tool was shown which was able to synthesize strategies for games with a moderate amount of weighted configurations. Additionally, a language for expressing games was presented, named LEG (Language of Energy Games). The formal syntax and semantics of LEG were also given given.

In this report we give refined theoretical definitions of energy games, as well as a discretely timed variant of energy games. We elaborate on how a strategy can allow a system to run infinitely, if such a strategy is possible, and we give a logical explanation of how this strategy can be synthesized.

We also present a user's guide to using LEG to express energy games in EgGS. While doing so, we also present how a discretely timed game can be simulated, using only the existing syntax of LEG.

As a follow-up to the results from our previous report, we present new methods which allow EgGS to handle much larger games, and synthesize strategies for these within much less time, than the former explicit approach. We will go into detail of how we obtain a symbolic representation of energy games, and how this symbolic representation is used algorithmically to synthesize strategies. While doing so, we show in detail how transitions are encoded using quantified boolean expressions.

Following this, we present experimental data comparing the symbolic approach to the explicit approach, and comment on the performance of both approaches. This is followed by a discussion of the results, and the established complexities of the symbolic representation. Finally, we conclude the report, by summing up the established results, and explain the impact of the achieved results, in comparison to the previous inefficient approach.

Mads Vestergaard Carlsen
mcarls08@student.aau.dk

Rasmus Søgaard Jacobsen
rjacob09@student.aau.dk

CONTENTS

CHAPTER 1

INTRODUCTION

In this report we present EgGS - the Energy Game Strategizer. EgGS is a tool for synthesizing strategies for energy-critical systems.

In this tool we can accurately and concisely express energy games using the syntax of LEG (Language of Energy Games). We specify energy games between an environment and a controller. In particular, the tool can synthesize a strategy for the controller (if any such strategy exists), which will allow the game to proceed indefinitely without exceeding some specified energy bounds. In Figure 1.1, we see an example of specifying a game in EgGS GUI.

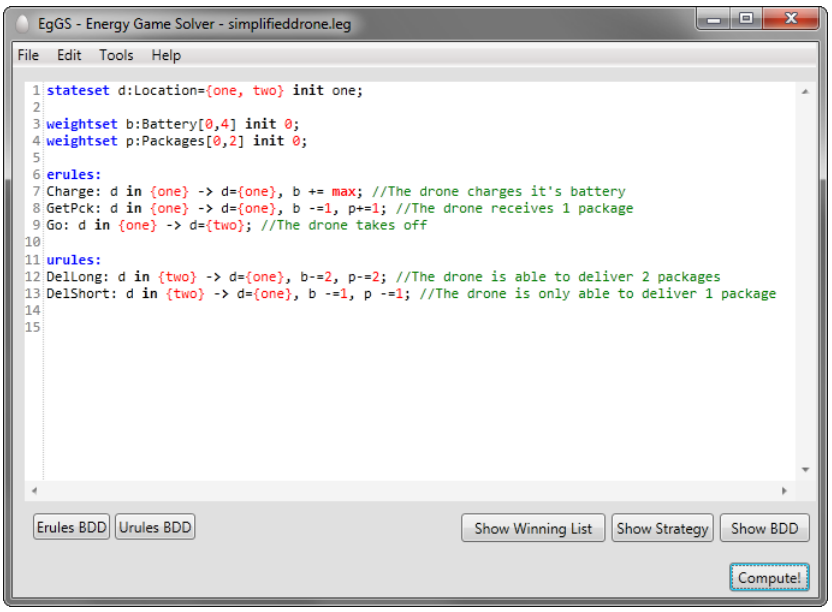


Figure 1.1: EgGS in use.

Seen here is the window for specifying the game itself using LEG. After computing the winning

states of the game, we may choose to view the winning states of the game, or a strategy for the controller. We will elaborate further on the features of EgGS in Chapter 6.

1.1 Motivation

There is an ever growing market and number of applications for energy efficient and autonomously functioning systems.
In distributed, embedded and reactive systems, energy management is important for ensuring the most efficient operating parameters for those systems. These devices vary from many fields, both private, corporate or for research. It impacts our daily lives in ways hidden to most, to a great degree. Some devices may influence our daily lives in a more direct manner, and we are continuing to see more use of such systems.

A recent example of this is flying drone package delivery. Major online retailers in the U.S. and China have begun to experiment with delivering packages by autonomous flying drones. These drones will deliver packages, ordered online, autonomously in urban areas. They are entirely autonomous and as such are also subject to energy bounds and other resource constraints. In fact entire packaging centers for such retailers may be automated almost entirely. In complex systems such as these, it is of great importance to the efficiency of the system that the devices can function in appropriate periods of time, to avoid wasting both time and energy.
For example, a robot may return to charge too early, while it still may be able to perform some small task. By optimizing this with a strategy, other resources can be saved, such as the amount of time spent in a charging station, thereby requiring fewer charging stations for a specific number of robots. Furthermore it may be possible to maximize the efficiency of the battery capacity by performing a specific sequence of actions which may not be immediately obvious.

Another example of this is small autonomous devices for private use, such as autonomous vacuum cleaners or lawnmowers. These devices are also subject to several energy constraints, and may also save money or time by following a certain strategy. For example, a robotic vacuum cleaner may decide to clean different parts of the house, depending on the time of day, the type of flooring, or the current battery capacity, whilst ensuring that it does this job without running out of battery, thus being able to function indefinitely.

It is the goal of all these autonomous systems to function to their highest efficiency, and operate indefinitely. In the scope of this project we are considering critical components, which cannot easily be salvaged in case they fail. The problem which inspired the work was energy management of miniaturized satellites known as CubeSATs.
It is both challenging, impractical and expensive to retrieve such devices in case of failure. These devices are great examples of energy critical systems which require a specific and tightly bound energy strategy.
Systems such as these are subject to a large number of variables in automated operation, in energy, time and any other resource constraint. The design of the system is not only greatly important, but also the decision-making process of which actions to take given a specific situation. These designs need to be thoroughly examined and verified before deployment,

which can be extremely challenging for complex systems. Synthesizing an energy strategy will ensure operational safety in such energy critical systems.

1.2 Contribution

- We define a formalism for Multiweighted Energy Games, for use in the context of a tool for synthesizing energy aware strategies.
- We introduce the language LEG (Language of Energy Games). LEG can accurately express large games by creating states, combinations of states and rules (transitions). The syntax and semantics are defined.
- We present methods used for synthesizing strategies for the controller, with a symbolic representation. The symbolic representation allows for synthesis in a much shorter time than through an explicit approach.
- We present the tool EgGS (Energy Game Strategizer) - a tool which allows us to express energy games in LEG and synthesize winning strategies. A description of the implementation and features of EgGS is given.

1.3 Related Work

In this report we continue the work done in a previous project. In the previous project [8], we have shown how Energy Games can be expressed using an expressive language named LEG (Language of Energy Games). We have also presented a prototype implementation of a tool which compiles LEG syntax into an energy game. The main focus of the previous project was the development of the LEG language, and the groundwork for implementation.

Energy games with lower and weak upper bounds with a single weight were presented in [5] and studied in [12, 11, 10] for single-weighted games, and complexities established herein. Multiweighted energy games have been studied in [6] and [9] for unary encoding.

Timed games were discussed in [5], where an undecidability result was shown for timed weighted energy games. Timed games were also briefly mentioned in [13], where lower and upper complexity bounds for several variants of energy games are also given. In this report we study energy games similar to [13], however herein we do not consider energy games turn-based, and players do not have ownership over any states, rather players have ownership over transitions.

CHAPTER 2

MULTIWEIGHTED ENERGY GAMES

In this section we will formally define Energy Games. Some parts of this chapter are excerpts and rewritten passages from our previous report[8] on this subject.

2.1 Introduction to Energy Games

Energy Games are two-player games played on a finite integer-weighted graph. In this game it is the objective to find a strategy that ensures infinite runs are possible while accumulated weights stay within lower and/or upper bounds. These bounds may be either weak or strong depending on the type of game.
We say that these games are multiweighted *ie.* the weights can have arbitrary dimensions. To introduce energy games we present a simple example, a game with strong upper and lower bounds, seen in Figure 2.1.

In this example we are simulating a flying drone. This drone delivers packages for an online store. This drone is subject to two different bounds, and as such each transition label in the energy game is a two-dimensional vector. In this case the resources represented by this vector is the battery capacity b and the package capacity p. We will refer to this weight-vector as (b, p).
The upper bounds of the energy game are represented by b_{max} and p_{max}. Energy Games may be subject to different bounds, and different types of bounds. If bounds are weak, then we may be able to perform an action which would accumulate a value on a specific weight which is higher than the bound, but the value would stay capped at the bounds. If it is strongly bound, then no action is permitted that would break the bound. Note that a game with both weak upper and lower bounds is trivially true. The game is played between a controller and the environment. These players are also referred to as the *existential player* (the controller) and the *universal player* (the environment). Intuitively - the existential player has the free choice when defining a strategy for the game, and the universal player, our opponent, controls all other actions. In the context of our drone example, the universal player chooses whether to

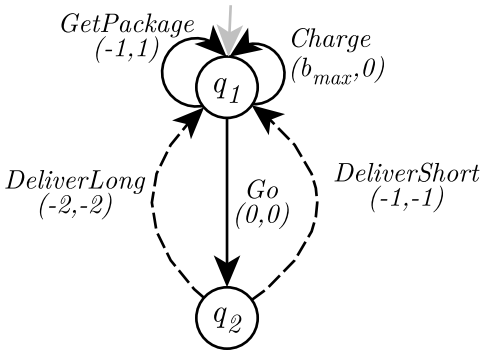


Figure 2.1: A flying drone example.

deliver one or two packages, representing whether or not the two delivery addresses are close enough to each other to perform two deliveries. Transitions that are shown with regular lines are owned by the controller, and those shown with dashed lines are owned by the environment. This means that any given node with a dashed transition will allow the universal player to have the first choice, either pick a universal transition, or force the existential player to pick an existential transition.

Definition 2.1 *A k-weighted energy game G is a four-tuple.*

$$G = (\mathbf{Q}, \Longrightarrow_{\exists}, \Longrightarrow_{\forall}, q_0)$$

where $\mathbf{Q}$ *is a finite set of configurations,* $q_0 \in \mathbf{Q}$ *is the starting configuration,* $\Longrightarrow_{\exists}$ *and* $\Longrightarrow_{\forall}$ *are the transition rules. The transition rules* $\Longrightarrow_{\Gamma}$ *are finite subsets of:*

$$\Longrightarrow_{\Gamma} \subseteq \mathbf{Q} \times \mathbb{Z}^k \times \mathbf{Q}$$

where $\Gamma = \{\exists, \forall\}$.

Each transition in a k-weighted energy game has a k-dimensional integer-vector cost. In the example in Figure 2.1 that vector has a dimension of two. We denote by $\mathbb{Z}^k$ the set of integer vectors of dimension $k > 0$. A weight cost vector is denoted $\overline{w}$ and $\overline{w}[i]$ is the i'th coordinate of that vector. The accumulated weight is denoted by the vector $\overline{v}$.
We say that a vector $\overline{w} \leq \overline{v}$ iff $\overline{w}[i] \leq \overline{v}[i]$ for all i where $1 \leq i \leq k$.

2.1.1 Semantics

A semantic energy game is played in a directed acyclic graph, $\mathcal{G} = (Q \times \mathbb{Z}^k, \longrightarrow_{\exists}, \longrightarrow_{\forall})$.
The transition relations in $\mathcal{G}$ are given by the transition rules:

$$(q, \overline{v}) \longrightarrow_\Gamma (q', \overline{v} + \overline{w}) \text{ in } \mathcal{G} \Leftrightarrow q \overset{\overline{w}}{\Longrightarrow}_\Gamma q' \text{ in } G, \Gamma = \{\exists, \forall\}$$

where q is the source configuration, $\overline{w}$ the weight vector, and q' is the target configuration.
We write $(q, \overline{v}) \longrightarrow_\Gamma (q', \overline{v'})$ whenever $(q, \overline{w}, q') \in \longrightarrow_\Gamma$ where $\overline{v'} = \overline{v} + \overline{w}$.
For every rule describing a transition in an energy game, there exists a corresponding set of transitions $q \xrightarrow{\overline{w}}_\Gamma q'$ between weighted configurations $((q, \overline{v}) \in \mathcal{G}$.

Each transition belongs to the controller ($\exists$) or the environment ($\forall$). When a weighted configuration has both universal and existential outgoing edges the universal player can choose a universal transition, or force the controller to choose an action.

When talking about energy games, and these types of systems in general, we are interested in systems that are non-terminating. In the context of energy games, this means that we are only interested in infinite runs, to keep the system running indefinitely within certain resource constraints. For an energy game to be considered valid, we implicitly assume that every configuration in $\mathcal{G}$ is *non-blocking*, *i.e.* for every $(q, \overline{v}) \in \mathcal{G}$ there exists a transition $(q, \overline{v}) \xrightarrow{\overline{w}}_\Gamma (q', \overline{v'})$. In other words, for an energy game to be considered valid, every configuration in the graph has at least one outgoing transition.

A *run* is a sequence of transitions between weighted configurations, made from the different choices of transitions by the existential and the universal player from a given starting weighted configuration. This starting point is defined by a configuration being marked as the starting configuration, and all weights initialized to a certain value $\overline{v}$.
Our goal with these energy games is to synthesize a strategy which guarantees an infinite run. In a run of the game, it is the goal of the controller to achieve an infinite sequence. The transitions that lead to this sequence of weighted configurations are chosen by the controller and the environment.
An infinite run with weights is called a *k-weighted run.* This infinite sequence takes the form of

$$\rho = (q_0, \overline{v}_0), (q_1, \overline{v}_1), (q_2, \overline{v}_2), ...$$

where $q_j \in \mathbf{Q}$, $\overline{v}_0 = \overline{l}$, and $\overline{v}_{j+1} = \overline{v}_j + \overline{w} \in \mathbb{Z}^k$ where $\overline{w}$ is in some transition $(q_j, \overline{w}, q_{j+1}) \in \longrightarrow_\Gamma$.
A run is strongly or weakly bound on either the lower or upper bound, or both. These bounds are given by vectors of lower and upper bounds, denoted by $\overline{l}, \overline{b} \in \mathbb{N}_0^k$ respectively.
For all i in $\overline{v}$, the lower bound for $\overline{v}[i] = \overline{l}[i]$, and the upper bound for $\overline{v}[i] = \overline{b}[i]$. A full definition of strong and weak bounds and the variants they give rise to is given in Section 2.4. A run ρ is *winning* if for every transition in the run, $\rho = (q_0, \overline{v}_0), ..., (q_n, \overline{v}_n), ...$, it holds that $\overline{l}[i] \leq \overline{v}[i] \leq \overline{b}[i]$, in the case of the game being strongly upper and lower bound. A list of bound- and game types is given in Section 2.4.

Therefore, in an infinite run in a strongly bound game, it is required that $q_j \xrightarrow{\overline{w}_j} q_{j+1}$ where for every $j \geq 0$ it holds that $\overline{v}_{j+1}[i] \leq \overline{b}[i]$ for every i. Or in other words, every vector coordinate must be below the bounds for each i'th coordinate, in every step of the game. This is called a *k-weighted run restricted to strong upper- and lower bounds.*

A prefix of such a weighted infinite run is shown in Figure 2.2a. We will refer to the set of all weighted runs in G restricted to $\overline{b}$ and $\overline{l}$ as $WR_{\overline{bl}}(G)$.

2.2 Strategies

In this section we will formally define a strategy, which consists of a map σ from a weighted configuration, from a weighted run $\rho \in WR_{\overline{bl}}(G)$, to a next configuration $\sigma(q, \overline{v}) \in Q$. We say that a weighted run

$$\rho = (q_0, \overline{v}_0), ...(q_n, \overline{v}_n)(q_{n+1}, \overline{v}_{n+1}, ...)$$

respects a strategy σ if for all n either

$$q_n \xrightarrow{\overline{w}}_{\forall} q_{n+1}, \text{ with } \overline{v}_{n+1} = \overline{v}_n + \overline{w}$$
$$\text{or}$$
$$q_n \xrightarrow{\overline{w}}_{\exists} q_{n+1} \text{ with } q_{n+1} = \sigma(q_n, \overline{v}_n)$$

If a weighted run respects a strategy, then we know that every weighted configuration in that run is *winning*. σ is winning in $(q, \overline{v})$ if any run ρ starting in $(q, \overline{v})$, which respects σ is winning. In this case case, we say that $(q, \overline{v})$ is winning. If a winning run respects a strategy, we say that the strategy is winning.

In the example on Figure 2.1, at the starting configuration, assuming starting weights of $(0, 0)$, the drone is faced with a number of choices. In this example, we will show a winning strategy. An example of a winning strategy, given in the form of a strategy readable to humans, is given in Figure 2.2b.
The first obvious choice is to charge the battery. The variable b_{max} represents the maximum battery capacity. The charge transition adds the value of b_{max} to $\overline{v}[i]$ where i is the coordinate for b, in this case 0. Charging in any configuration where the battery is not exactly 0 is a losing move. This is due to the game being strongly bound. A winning strategy does exist for the given example, for some bounds. We can show that a strategy exists for the bounds $b_{max} = 4$ and $p_{max} = 2$. It does however not exist for *e.g.* $b_{max} = 3, p_{max} = 2$. Intuitively, the low battery capacity of 3 is not enough to make a long delivery, even though the drone can attempt that action. A strategy for a system specifies which action to take given a particular situation. We will now give a formal definition of a strategy.

Definition 2.2 *Memory-Less Existential Strategy*

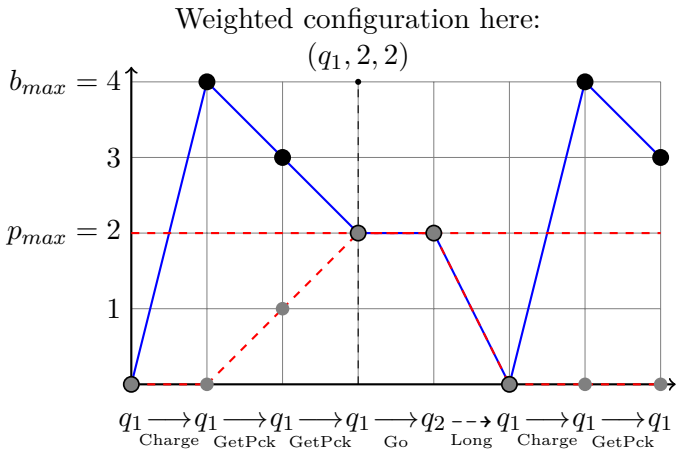
A strategy for the controller σ for a k-weighted game $G = (\mathbf{Q}, \longrightarrow_{\exists}, \longrightarrow_{\forall}, q_0)$ is a map from $\mathbf{Q} \times \mathbb{Z}^k$ to $\mathbf{Q}$, i.e.:

$$\sigma : \mathbf{Q} \times \mathbb{Z}^k \longrightarrow \mathbf{Q}$$

such that for some $\overline{w} \in \mathbb{Z}^k$, $(q, \overline{w}, \sigma(q, \overline{v})) \in \longrightarrow_{\exists}$.

When looking at Figure 2.2a we see a sample run of the game. Here we are looking at a prefix of an infinite run, to demonstrate the winning strategy of the controller.

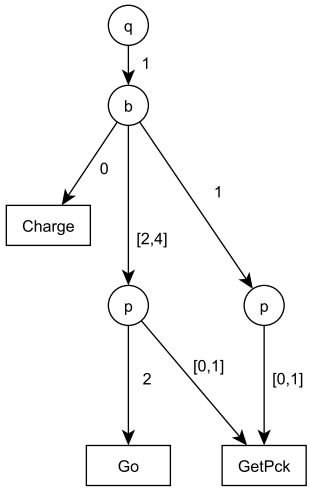
The blue (solid) line represents the current battery level, and the red (dashed) line represents the number of packages held. The bounds in this game are strong, and the lower bound is zero for both weights. The upper bound for the battery $b_{max} = 4$ as indicated on the graph, and the package capacity $p_{max} = 2$.



(a) A sample run prefix.

```
1 if configuration == q1
2   if b==0
3     Charge
4   if b>1 and p==2
5     Go
6   if b>0 and p<2
7     GetPackage
```

(b) Strategy in textual form.



(c) Strategy as MTIDD.

Figure 2.2: Sample run with a winning strategy, presented in textual form, and MTIDD form.

As a sample run consider starting from the initial weighted configuration of $(q_1, 0, 0)$. The first choice is to charge the battery, bringing us to $(q_1, 4, 0)$. From here it is possible to choose from any of the other actions, such as going on a delivery. However, depending on the universal player, either one or more packages will be delivered. Therefore, it is not possible to have a winning strategy that takes the *Go* transition while carrying less than two packages. In that case the universal player could choose the *DeliverLong* which would break the lower bound.

Likewise, the lower battery bound would be broken as well, if the deliver transition is taken with a battery level less than two. Therefore it should be apparent that since we cannot enter with less than two packages, and less than 4 in battery level, that a battery capacity of 3 would not allow a winning strategy - the battery capacity would be too low to fetch both packages and then go on a delivery.

This sample run seems to indicate that an infinite run is possible, and it is. This strategy has been synthesized by EgGS, and in the following Section 2.3 we will show how to synthesize a strategy. Given here is the winning strategy in two forms; in Figure 2.2b the strategy is specified textually. Strategies and their information might be expressed by MTIDDs as seen in Figure 2.2c, or textually as conditional statements as seen in Figure 2.2b.
An MTIDD is a Multi-Terminal Interval Decision Diagram, these are interval representations of binary decision diagrams, which are discussed in Section 4.1. However, the strategy should be apparent intuitively - from the first node, we check whether the configuration is 1 (representing q_1. The next node is an integer test of the battery value. Indicated by intervals, we check certain other values to determine which action to take, such as charging if b is 0, or proceeding to checking the number of packages, if b is in the interval $[2, 4]$.

2.3 Strategy Synthesis

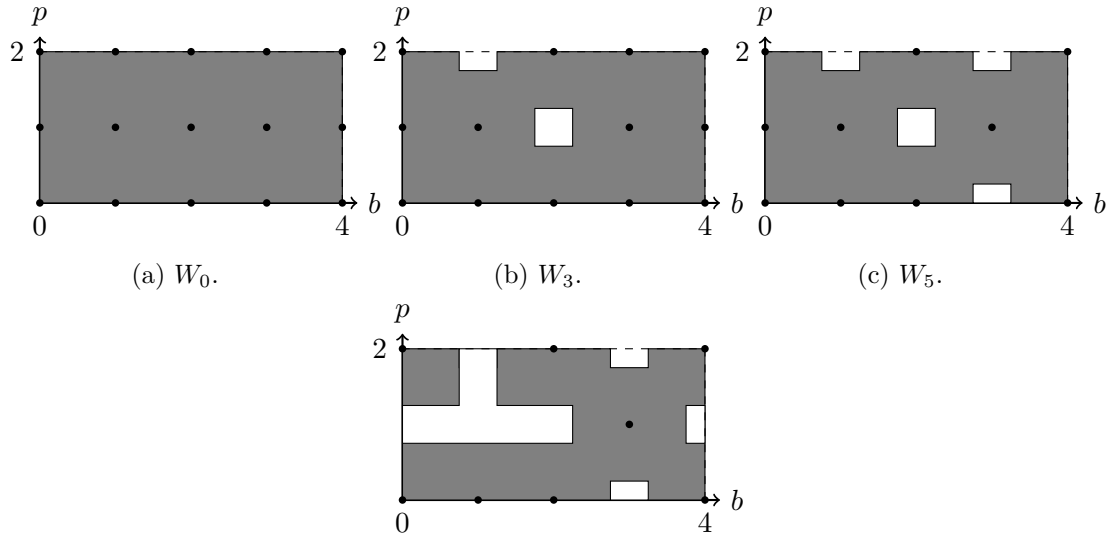
To synthesize a strategy, we mean to find a *winning space*. A configuration space, is a finite set of weighted configurations $\mathbf{W} = \{(q_0, \overline{v}_0), ..., (q_n, \overline{v}_n)\}$. For a configuration space to be considered winning, it must hold that $\overline{l} \leq \overline{v}_n \leq \overline{b}$ for any n, it holds that for all universal transitions, and for some existential transition with source $(q_n, \overline{v}_n)$, the target weighted configuration is also in the configuration space. This is called the *winning space*.
In this section we will elaborate on finding the maximal winning space as a fixed point. From this winning space, we can synthesize a winning strategy. If no such winning space exists, then no winning strategy is possible, and as such an infinite run is not possible for the game. A full definition of the fixed-point theorem and how it relates to this computation of a fixed point is given in Appendix C.
To briefly explain the algorithm used to find the winning space, we define a monotonic function which takes us from one winning space to the next, by applying the transitions of the game. In doing so we reduce the configuration space, by eliminating configurations which cannot reach a winning configuration by any transition from the previous configuration space. This is repeated until we reach the greatest fixed point, *i.e.* until $\mathbf{W} = F(\mathbf{W})$. The function $F : \mathbf{W} \to \mathbf{W}$ is defined as:

$$\begin{aligned}(q, \overline{v}) \in F(\mathbf{W}) \Leftrightarrow \quad & \overline{l} \leq \overline{v} \leq \overline{b} \\ & \wedge \\ & \forall (q, \overline{v}) \longrightarrow_{\forall} (q', \overline{v}') \Rightarrow (q', \overline{v}') \in \mathbf{W} \\ & \wedge \\ & \exists (q, \overline{v}) \longrightarrow_{\exists} (q', \overline{v}') \wedge (q', \overline{v}') \in \mathbf{W}\end{aligned} \tag{2.1}$$

$F(\mathbf{W})$ can be intuitively explained as follows: For every universal transition, the target weighted configuration of that transition must exist in $\mathbf{W}$, and for some existential transition, the target weighted configuration exists in $\mathbf{W}$.
The initial assumed winning space $\mathbf{W_0}$, is the set of all weighted configurations within the bounds of the game *i.e.* $(q, \overline{v}) \in \mathbf{W_0}$ for all q and all $\overline{v}$ where $\overline{l} \leq \overline{v} \leq \overline{b}$.
To achieve the greatest fixed point, the result of F is inputted to F again, and continues these iterations until the result is the same as the input.

An example of the application of F is illustrated in Figure 2.3. Each of the graphs represents some W_n for the drone example. In this case we are only showing the weighted configurations pertaining to the state q_1. As shown in Appendix C, we know that this function will reach a greatest fixed point and terminate eventually. Should this greatest fixed point be $\emptyset$, then the game is not capable of achieving an infinite run.

An example of the fixed point algorithm is shown in Figure 2.3. In this example we see



(a) W_0. (b) W_3. (c) W_5.

(d) W_8, the last iteration.

Figure 2.3: Fixed point iterations for the q_1 configuration of the drone example.

the application of F on the drone example, however, for simplicity only the weights of q_1 is shown. Since the controller only has control in this configuration, it is sufficient to consider the winning weights of this configuration.
In Figure 2.3a we see W_0. Here we have assumed every combination of weighted configurations to be winning. After two iterations, in Figure 2.3b, the algorithm has discovered two weighted configurations that are in fact not winning. This is because no transition exists which would not break the bounds of the game. After two more iterations, as seen in Figure 2.3d, more weighted configurations have been excluded - the transitions in those weighted configurations were leading to what was excluded in the previous iterations. This procedure continues until no further weighted configurations are excluded. When this point is reached, we have found a winning space of weighted configurations.

The winning space directly points to a strategy in itself - since the transition relation is known the controller can select a transition which leads to a weighted configuration in $\mathbf{W}$. One or more maps can also be algorithmically constructed from the winning space.

The main algorithm can be seen in Algorithm 1, and $F(\mathbf{W})$ in Algorithm 2. Some optimizations have been made on the function for some efficiency, such as limiting the possible configurations examined in each iteration to configurations in the last weighted configuration space.

Algorithm 1 Input: a game $\mathcal{G} = (Q, \longrightarrow_\exists, \longrightarrow_\forall, q0)$, and lower and upper bounds $\bar{l}, \bar{b}$.

```
1: W ← Q × {v̄ | l̄ ≤ v̄ ≤ b̄}
2: repeat
3:     W_old ← W
4:     W ← F(W_old)
5: until W = W_old
6: return W
```

Algorithm 2 $F(\mathbf{W})$.

```
1: W_new = ∅
2: bool valid
3: for all (q, v̄) ∈ W do
4:     valid = true
5:     for all (q, v̄) −→∀ (q′, v̄′) do
6:         if (q, v̄) ∉ W then
7:             valid ← false
8:             break for
9:     if valid then
10:        if (q, v̄) ↛∃ (q′, v̄′) then
11:            W_new ← W_new ∪ (q, v̄)
12:        else
13:            for all (q, v̄) −→∃ (q′, v̄′) do
14:                if (q′, v̄′) ∈ W then
15:                    W_new = W_new ∪ (q, v̄)
16:                    break for
    return W_new
```

2.4 Variants of Energy Games

As we previously mentioned, games are subjected to Lower and Upper bounds. These bounds can be either weak or strong, and in this section we will elaborate on bounds, and thereby the different variants of energy games. A bound is weak, if transitions which break the bounds are allowed, but when adding it to $\overline{v}[i]$, the value of $\overline{w}[i]$ is truncated, such that $\overline{v}[i] = \overline{b}[i]$ if the upper bound is broken, or $\overline{v}[i] = \overline{b}[i]$ if the lower bound is broken. Strong bounds (or just referred to as bounds), do not allow this truncation, and as such an action that breaks the bound is not a possible choice in a strategy. These differences mean a number of different variants of energy games, which we will summarize in this section.

k-Weighted Energy Games with Lower Bound (GL)

In this problem we ask whether the controller has a winning strategy, where all coordinates in $\overline{v}_i \geq \bar{l}_i$, in any infinite run following this strategy, *i.e.*:

For a game G and a vector of lower bounds $\bar{l} \in \mathbb{N}_0^k$, does there exists a strategy σ for the

controller, such that any weighted run $(q_0, \overline{v}_0), (q_1, \overline{v}_1), ... \in WR_{\overline{\infty}}(G)$ which respects σ, and satisfies $\overline{v}_i \geq \overline{l}$?

This problem has been shown to be no less than EXPTIME-hard but no more than k-EXPTIME in [13].

k-Weighted Energy Games with Lower and Weak upper bound (GLW)

Following the same rules as above in GL, but with the addition of a weak upper bound, such that for any coordinate that breaks the upper bound, that coordinate is truncated to $\overline{b}[i]$:

For a game G and two vectors of lower and upper bounds $\overline{l}, \overline{b} \in \mathbb{N}_0^k$ is there a strategy σ such that any weighted run in $WR_{\overline{\infty}}(G)$ respecting σ satisfies $\overline{l} \leq \overline{v}_i \leq \overline{b}$, where if $\overline{v}[i] > \overline{b}[i]$ then $\overline{v}[i] = \overline{b}[i]$?

This problem was shown to be polynomial time reducible to GLU in [13].

k-Weighted Energy Games with Lower and Upper bound (GLU)

We ask whether, given a game and two vectors of lower and upper bounds, does there exist a strategy for the controller in which all accumulated weights stay above the lower bound and stays at a value below or equal to the upper bounds?

For a game G, and two vectors of lower and upper bounds $\overline{l}, \overline{b} \in \mathbb{N}_0^k$, is there a strategy σ such that any weighted run $(q_0, \overline{v}_0), (q_1, \overline{v}_1), ... \in WR_{\overline{bl}}(G)$ which respects σ satisfies $\overline{l}_i \leq \overline{v}_i \leq \overline{b}_i$?

This problem was shown to be polynomial time reducible to GLU(1) (*i.e.* energy games with a single weight coordinate, as seen in [5], and as such is EXPTIME-complete.

Other specializations

We can describe some specializations of other problems of energy games, such as considering a game where there exists only existential choices *i.e.* $\longrightarrow_\forall = \emptyset$ (the *existential variant*) or likewise, a universal-only game where $\longrightarrow_\exists = \emptyset$ (the *universal variant*).

For the existential variant it is sufficient to ask whether some weighted run *exists* within given bounds. However, for the universal variant it is required that *all* weighted runs have accumulated weights within the given bounds.

We also introduce further possibilities for updating the weights of the configuration later in the definition of LEG. In LEG we allow weight updates to be dependent on arithmetic expressions which rely on the configuration. As explained in Section A.2, the construction of energy games through LEG consists of weighted configurations for all sets. Knowing this, it is possible to define vector updates as $\overline{w}[i] = \overline{v}[i] * 2$ and other similar arithmetic expressions.

In other sources, energy games are expressed as turn-based, with the controller or the environment having ownership over configurations rather than transitions. In Appendix B, it is shown that the approach considered in this report is reducible to turn-based games.

2.5 Discretely Timed k-weighted Energy Games

It is an obvious thing to ask for an extension with time to energy games. In this section we will give definitions for discretely timed energy games.
In Section 3.3, we will show how such a game can be simulated using our language for expressing energy games, LEG, with minor limitations. Since energy games with continuous time have been shown to be undecidable even for one clock in [5], we here consider a discrete setting, where time is represented as integer time units.
A Discretely Timed Energy Game is a k-weighted Game played in discretely timed priced automata with energy constraints. As with regular energy games, transitions belong to either the controller or the environment.

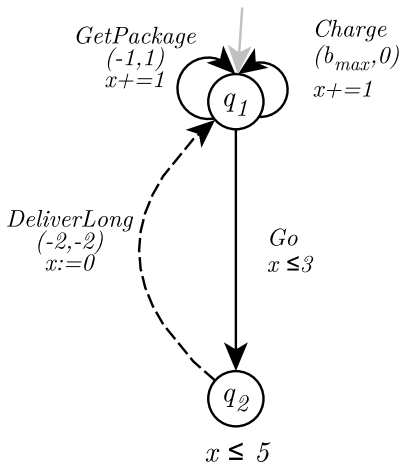


Figure 2.4: Simplified timed drone.

Clocks in a timed game must satisfy any guard on transitions. By $\mathcal{B}(\mathbf{X})$, we denote the set of boolean clock constraints (guards) over the set of clocks $\mathbf{X}$.
These constraints are a subset of boolean logic, given by the syntax $g ::= x \bowtie n \mid g1 \wedge g2$, where $\bowtie = \{\geq, >, \leq, <, =\}$.

Definition 2.3 *Discretely Timed k-weighted Energy Game*
A discretely timed k-weighted energy game is a 6-tuple

$$G_{\mathcal{T}} = (\mathbf{Q}, \mathbf{X}, I, \Longrightarrow_{\exists}, \Longrightarrow_{\forall}, q_0)$$

Where $\mathbf{Q}$ *is a finite set of configurations,* $q_0 \in \mathbf{Q}$ *is the starting configuration, and* $\mathbf{X}$ *is a finite set of clocks of* $G_{\mathcal{T}}$.
$I : \mathbf{Q} \to \mathcal{B}(\mathbf{X})$ *assigns invariants to configurations.*
Our transition rules $\Longrightarrow_\Gamma$ *are given by:*

$$\Longrightarrow_\Gamma \subseteq \mathbf{Q} \times \mathbb{Z}^k \times \mathcal{B}(\mathbf{X}) \times \mathbf{2}^{\mathbf{X}} \times \mathbb{N} \times \mathbf{Q}$$

where $\Gamma = \{\exists, \forall\}$. *The game forms transitions between timed weighted configurations* $(q, \overline{c}, \overline{v})$ *where* $\overline{c} \in \mathbb{N}^{|\mathbf{X}|}$. *Timed transitions are written* $q \xrightarrow[d]{\overline{w}, r, g,} q'$ *whenever* $(q, \overline{w}, g, r, d, q') \in \longrightarrow$ *where* r *is the set of clocks to reset,* g *the set of guards,* d *the duration, and* $\overline{c}$ *the clock values. As with non-timed games* q *is the source configuration and* q' *the target configuration.*

This effectively means that in a timed energy game, a transition is only possible if the guards g are satisfied for every clock value $\overline{c}$.

As an example consider a smaller version of the previously given drone example, with a single clock x as seen in Figure 2.4. In this example, the drone is able to charge and load up packages as normal, however in this case, there is a constraint on how fast it must do these things, if this is not done in three time units, the package is too delayed, and it is not possible for the drone to make a delivery. Here it is assumed that each action consumes one time unit, such as charging or retrieving a package. As such, the drone must not delay at all before going on a delivery, or the guard cannot be satisfied, and the game is lost, since no transition in q_1 is possible without breaking the bounds.
In q_2, the universal player may force the existential player to delay, but when the invariant is reached, even the universal player is forced to make his move. After the delivery is done the clock is reset. This game can run infinitely[1].
In Section 3.3, we will show how this game can be encoded using untimed energy games in LEG.

2.5.1 Semantics

A semantic discretely timed k-weighted energy game is played in a directed acyclic graph $\mathcal{G}_{\mathcal{T}} = (Q \times (\mathbf{X} \to \mathbb{N}) \times \mathbb{Z}^k, \longrightarrow_\exists, \longrightarrow_\forall)$.
The transition relations are given by the transition rules:

$$(q, \overline{c}, \overline{v}) \longrightarrow_\Gamma (q', \overline{c}', \overline{v} + \overline{w}) \text{ in } \mathcal{G}_{\mathcal{T}}$$
$$\Leftrightarrow$$
$$q \xrightarrow[d]{\overline{w}, g, r} q' \text{ in } G_{\mathcal{T}}.$$

For which it holds that $\overline{c} \models g$, $\overline{c}' \models I(g)$ and $\overline{c}' = \overline{c + d}[r]$.
Additionally, the transition relations have for every configuration in Q the delay transition $\xrightarrow{1}$:

$$(q, \overline{c}, \overline{v}) \xrightarrow{1}_\exists (q, \overline{c} + 1, \overline{v}) \text{ where } \overline{c} + 1 \models I(q).$$

[1] This was verified by EgGS

$\overline{c}+1 \in \mathbb{N}^{|\mathbf{X}|}$ is given as $(\overline{c}+1)(x) = \overline{c}(x)+1$ for all $x \in \mathbf{X}$.

An infinite run with time is called a *timed k-weighted run.* This infinite sequence takes the form of

$$\rho = (q_0, \overline{c}_0, \overline{v}_0), (q_1, \overline{v}_1, \overline{v}_1), (q_2, \overline{c}_2, \overline{v}_2), ...$$

where $q_j \in \mathbf{Q}$ and $\overline{v}_{j+1} = \overline{v}_j + \overline{w} \in \mathbb{Z}^k$ where $\overline{w}$ is in some transition $(q_j, \overline{w}, g, r, d, q_{j+1}) \in \longrightarrow_\Gamma$.

2.6 Timed Strategy

A strategy in a timed game is specified similarly to a strategy in a non-timed game. In non-timed games, decision are made depending on the configuration and the weights of the resource vector. In the case of timed games, we consider the value of the clock(s) as well. A strategy for the timed drone example is given in Figure 2.5. A strategy must stay within all invariants and clock guards. It should be noted, that in an energy game, both the existential and universal player must satisfy invariants.

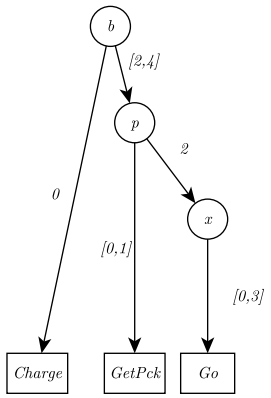
Definition 2.4 *Timed Existential Strategy*
A strategy for Player 1, $\sigma_\mathcal{T}$, is a strategy for a game $G_\mathcal{T}$. It is a map from a timed weighted configuration i.e.:

$$\sigma_\mathcal{T} : \mathbf{Q} \times \mathbf{X} \to \mathbb{N} \times \mathbb{Z}^k \longrightarrow (\{1\} \cup \Longrightarrow_\exists)$$

such that if $\sigma_\mathcal{T}(q, \overline{c}, \overline{v}) = (q', g, \overline{v}, r, d, q'')$, then $q' = q$, where $\overline{c} \models g$ and $\overline{c}[r] \models I(q'')$.

```
1 if configuration == q1
2   if b==0
3     Charge
4   if b>1 and p==2 and x <=3
5     Go
6   if b!=0 and p<2
7     GetPackage
```

(a) Textual strategy.



(b) MTIDD strategy.

Figure 2.5: Strategy for the timed drone in Figure 2.4.

We say that a timed weighted run

$$\rho = (q_0, \overline{c}_0, \overline{v}_0), ...(q_n, \overline{c}_n, \overline{v}_n)(q_{n+1}, \overline{c}_{n+1}, \overline{v}_{n+1}, ...)$$

respects a timed strategy $\sigma_\mathcal{T}$ if for all n either

$$q_n \xrightarrow[d]{\overline{w},g,r}_{\forall} q_{n+1}, \text{ with } \overline{v}_{n+1} = \overline{v}_n + \overline{w}$$
$$\text{or}$$
$$q_n \xrightarrow[d]{\overline{w},g,r}_{\exists} q_{n+1} \text{ with } (q_{n+1} = \sigma(q_n, \overline{c}_n, \overline{v}_n).$$

In a strongly bound timed game, a run ρ is winning if for every transition in the run, $\rho = (q_0, \overline{c}_0, \overline{v}_0), ..., (q_n, \overline{c}_n, \overline{v}_n), ...$, it holds that $\overline{l}[i] \leq \overline{v}[i] \leq \overline{b}[i]$, in the case of the game being strongly upper and lower bound. A timed strategy is winning, if for any winning timed weighted configuration $(q_n, \overline{c}_n, \overline{v}_n)$, $\sigma(q_n, \overline{c}_n, \overline{v}_n)$ is part of any winning run ρ.

2.7 Timed Strategy Synthesis

For solving these timed energy games, we adapt the algorithm shown in Section 2.3. We will adapt the algorithm to take clocks into consideration.
This is an adaptation of the function $F(\mathbf{W})$ which was shown in Equation 2.1.

$$\begin{aligned}(q, \overline{c}, \overline{v}) \in F(\mathbf{W}) \Leftrightarrow \quad & \overline{l} \leq \overline{v} \leq \overline{b} \\ & \wedge \\ & \forall (q, \overline{c}, \overline{v}) \longrightarrow_{\forall} (q', \overline{c}', \overline{v}') \Rightarrow (q', \overline{c}', \overline{v}') \in \mathbf{W} \\ & \wedge \\ & \Big(\exists (q, \overline{c}, \overline{v}) \longrightarrow_{\exists} (q', \overline{c}', \overline{v}') \wedge (q', \overline{c}', \overline{v}') \in \mathbf{W} \\ & \vee \\ & \exists (q, \overline{c}, \overline{v}) \xrightarrow{1} (q, \overline{c}+1, \overline{v}) \wedge (q, \overline{c}+1, \overline{v}) \in \mathbf{W} \Big)\end{aligned}$$

The main algorithm is still the same, however, here the value of the clocks is considered as well. This means that a configuration space in the timed algorithm is a finite set of timed weighted configurations, *i.e.* $\mathbf{W} = \{(q_0, \overline{c}_0, \overline{v}_0), ..., (q_n, \overline{c}_n, \overline{v}_n)\}$.
As an addition, a disjunction is added which states that if the timed weighted configuration resulting from a delay transition is in $\mathbf{W}$, then the third conjunction holds. The application of this fixed point algorithm would result in a three-dimensional graph, and as such is not pictured here.

CHAPTER 3

THE LEG LANGUAGE

In this chapter we give an introduction to the LEG language. It will also cover any additions to LEG, compared to the version of LEG presented in [8]. Additionally we will show how we can represent discretely timed games using the existing features of LEG, and show how LEG could be extended to cover guards as part of its own syntax as well.

3.1 Introduction to LEG

In this section we introduce the three elements that make up LEG: *statesets*, *weightsets*, and *rules*. We then proceed to show some of the games that can be expressed with LEG.

3.1.1 Statesets

Statesets are used to express the configurations of a game. All configurations in a game corresponds to a tuple of states - a state from each stateset. This combination of states, is the reasoning behind naming each vertex in the final graph a configuration. An example of a stateset declaration is given in Listing 3.1. This stateset declares the configurations `one` and `two`.

```
1 stateset x:X={one, two} init one;
```

Listing 3.1: Example of a stateset.

The keyword `stateset` declares that the following is a stateset declaration. The lower-case `x` is an identifier for the stateset variable. The stateset variable is used in rules to refer to states in configurations. These identifiers can consist of lower- and upper-case letters as well as numbers, but must begin with a lower-case letter. The upper-case `X` is the identifier of the stateset and it is equivalent to the full set of states in that stateset. The identifier of the stateset can, like the variable identifier, have a complex name, but must begin with an

upper-case letter. The set of states is embraced by brackets and contains a finite list of states. Each state can have a complex identifier, and follows the same rules as a variable identifier *i.e.* it must begin with a lower-case letter. Finally `init one` describes the initial state of the statevariable. These initial states must be marked in each stateset, as they mark the starting configuration of the game.

Given only one stateset, the configurations are equivalent to the states of that stateset. The expressive power of LEG becomes apparent when two or more statesets are declared. Observe the statesets declared in Listing 3.2. These two statesets describe configurations corresponding to the combinations of states from each stateset. The configurations described in Listing 3.2 are `(one,alpha)`, `(one,beta)`, `(two,alpha)`, `(two, beta)`. Adding a third state to either stateset would increase the number of configurations to 6. Adding a third stateset would increase the number of configurations as a multitude of the number of states declared in the third stateset. This can be of great assistance in the creation of large systems with many configurations. This feature may for example be used to model whether a system is in some state which is common for all states of the system. For example, a satellite may have a stateset describing whether or not it is currently exposed to sunlight.

```
1 stateset x:X={one, two} init one;
2 stateset y:Y={alpha, beta} init alpha;
```

Listing 3.2: Example of two statesets.

3.1.2 Weightsets

Weightsets are used to express the resources of a game. It shares some similarities with the stateset. An example of a weightset declaration can be seen in Listing 3.3.

```
1 weightset a:A[0,4] init 0;
```

Listing 3.3: Example of a weightset.

The keyword `weightset` declares that the following is a weightset declaration. As with statesets, the lower-case letter `a` is a variable and the upper-case `A` is the identifier of the weightset. The same naming syntax applies to variable- and weightset identifiers. While the syntax for a weightset is given as an interval, in this case `[0,4]`, the actual meaning is that of a set of integers. The set of integers includes every integer between the minimum value and the maximum value, both values included. Thus, the weightset `A` in Listing 3.3 contains the set of integers $0, 1, 2, 3, 4$.

Recall that an Energy Game has an accumulated weight vector, $\overline{v}$, and each transition has a vector weight. Each weightset defines the valid values of a coordinate in $\overline{v}$. A weight vector associated with some transition likewise has an entry for each coordinate in $\overline{v}$, *i.e.* a manipulation on the weightset. The indexation of this vector is determined by the order of declaration of weightsets. An index for the weightsets in Listing 3.4 would be `(a,b)`.

```
1 weightset a:A[0,4] init 0;
2 weightset b:B[0,1] init 0;
```

Listing 3.4: Example of two weightsets.

3.1.3 Rules

An semantic game has transition relations between each weighted configuration. Every transition has an associated cost on one or more weights. Any weight that is not mentioned, is implicitly assumed to have a 0-cost. As mentioned earlier, every transition is owned by either the controller, or the environment.

Rules in LEG, correspond to the transitions rules shown in Definition 2.1. Each rule defines a set of transitions between weighted configurations, and as such one rule may result in a multitude of transitions. Rules are either defined as being either universal or existential. An existential rule is embraced by chevrons, e.g. `<rule>` and universal rules are embraced by square brackets, e.g. `[rule]`. The common usage, however, is to declare a list of existential rules and a list of universal rules. The existential can be declared in a block by writing `erules:`, followed by any number of rules. The universal block is specified in the same manner, but writing `urules:` instead. This is shown in Listing 3.6

A rule consists of two primary elements, a condition, and an update. The condition comes before the keyword `->`, followed by the update. Observe the rule in Listing 3.5. The condition here is `x in {one}, y in {alpha}`, which results in the update `x={two}, b+=1;`. The condition of a rule only relates state variables, which are matched to any matching configuration. The update specifies the resulting configurations, by relating state variables, but may also specify a cost for a weight variable.

```
1 x in {one}, y in {alpha} -> x={two}, b+=1;
```

Listing 3.5: Example of a rule.

In the condition, a state variable may be specified to be in one or more states, e.g. `x in {one}` specifies that this rule is only enabled for configuration that includes the state `one`. The case of `x in { one, two}`, means that the rule is enabled for any configuration in which the state is either `one` or `two`. As `x in { one, two}` is equivalent to `x` being in any of its states, it is possible to write `x in X`[1] instead. It is also possible to exclude `x in X`, as this is equivalent to specifying "it does not matter what `x` is". The condition of a rule must always specify states for at least one state variable.

The update of a rule specifies the valid target configurations as well as the manipulation performed on accumulated weights. The target configurations are identified through state variables. Continuing the example in Listing 3.5, the update of the rule being `x=two, b+=1`, the resulting configuration may be any configuration in which `y` is in state `alpha` and `x` is in state `two`. If the state variable `x` was related to all the states, it could be written as `x=X`.

[1] Currently not supported in the symbolic algorithm.

Left-hand side	Assignment	Equation
		1
		$1 + 1$
		$1 - 1$
	$=$	$1 * 1$
`w`	$+=$	$1+$ `v`
	$-=$	$1-$ `v`
	$*=$	$1*$ `v`
		`max`
		`min`

Table 3.1: Weight assignments.

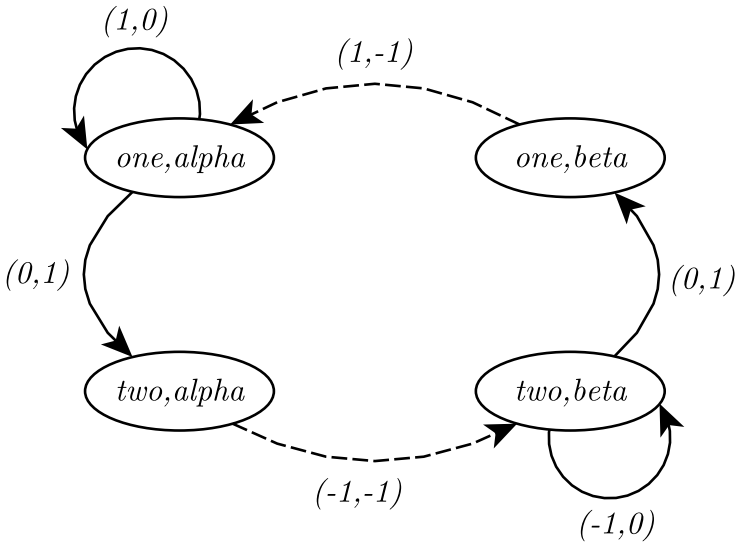


Figure 3.1: Example game.

Contrary to the condition of a rule, not writing a state variable in the update part of a rule is equivalent to no change to the state.

In the update of a rule, the manipulation of a weight, e.g. `b+=1`, is called a weight assignment. A basic assignment is performed by `=`, but the shorthands `+=`, `-=`, `*=` are equally correct syntax. The left-hand side of a weight assignment always relates to a weight variable. The right-hand side of a weight assignment can be either a simple or complex equation. An example of a simple equation could be a single integer, while a complex equation may contain weight variables[2]. It is also possible to use the keywords `max` and `min` in place of an equation - these keywords denote the upper or lower bound of the weight. The possible equations and assignments can be seen in Table 3.1, where `w` denotes a weight variable and `v` denotes any weight variable, `w` included.

[2]Currently not supported in the symbolic algorithm.

An example of LEG is given in Listing 3.6 and the corresponding graphical representation of the game is shown in Figure 3.1.

```
1  stateset x:X={one, two} init one;
2  stateset y:Y={alpha, beta} init alpha;
3
4  weightset a:A[0,4] init 0;
5  weightset b:B[0,1] init 0;
6
7  erules:
8  x in {one}, y in {alpha} -> a+=1;
9  x in {two}, y in {beta} -> a-=1;
10 x in {one}, y in {alpha} -> x={two}, b+=1;
11 x in {two}, y in {beta} -> x={one}, b+=1;
12
13 urules:
14 x in {two}, y in {alpha} -> y={beta}, a-=1, b-=1;
15 x in {one}, y in {beta} -> y={alpha}, a+=1, b-=1;
```

Listing 3.6: Declaration of an Energy Game.

3.2 Updates to LEG

There have been some updates to the syntax of LEG since [8]. These changes do not affect the semantics of LEG.

Comments are now possible in LEG. Comments are lines of text which are ignored by the lexer and parser. The beginning of a comment is marked by // and the comment ends with the end of line. This feature is beneficial for writing small comments about an LEG line, such as explaining what a rule does or what a stateset represents.

In weight assignments, some fine-tuning to the expression of equations has been done. The option to include parentheses in the equation has been added, regulating the order of evaluation. This allows for weight assignments such as `w+= 2*(v-2)`, which increases the expressiveness of LEG. Another change to the syntax of weight assignments is the option to use the keywords `max` and `min` as part of an equation. Previously it was only possible to encode lines such as `w+=max`, but now it is also possible to encode lines such as `w+=max-3`. With the previous implementation, the weight assignment `w+=max` would model a transition that would represent a valid move only when the accumulated weight was 0. With the new option, it is easier to model transitions that are restricted to values between 0 and higher, such as `w+=max-3`, which will only represent a valid move when the accumulated weight is between 0 and 3.

Finally, rules can be named. This has the syntax `Name: rule`, where `Name` is the name of a rule and `rule` is simply a substitute for a full rule. Names of rules must start with upper-case letters and end with colon.

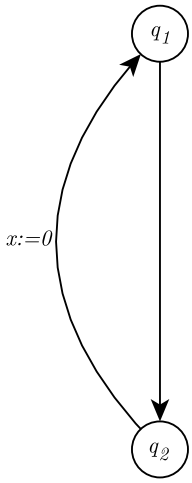
Examples of all these new features of LEG can be seen in the timed energy game from Figure 3.6, Listing 3.6b.

3.3 Timed Games in LEG

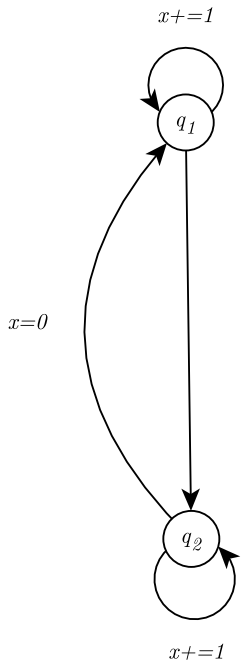
In this section we will show how all the features of discreetly timed automata can be simulated. Currently LEG has no built-in support for guards, clocks and invariants, but these can still be expressed through use of the current syntax. We will show how each of these are made.
In LEG, a clock can be expressed as a weightset, with the upper bound acting as a global invariant. By creating intermediate configurations which utilizes LEG syntax features, we can create any features necessary for this weightset to function as a clock. By associating a certain weight to each transition for the clock, we can add a time cost to every action in the game.

3.3.1 Delays

A timed game is able to delay at any point, as long as it satisfies any eventual invariants in that configuration, or any global invariants. By global invariants, we are referring to the bounds of the clock. Since we are simulating clocks as weightsets, clocks are bound as well. Assume that a global invariant means placing an invariant on all configurations in the game. As mentioned in Section 2.5 however, we are simulating a discretely timed game. Since all resources in LEG operate on integers, this is true here as well, and as such the least amount of delay is 1 time unit. In Figure 3.2 we can see how to simulate delays in LEG. It is possible



(a) Player 1 can delay at any time.



(b) Looping transitions to simulate delay transitions.

Figure 3.2: Simulating delay transitions with LEG.

for Player 1 to delay at any time when playing in all states of the abstract version seen in Figure 3.2a. So, to simulate the delays we simply add self-looping transitions which add the lowest possible time unit. This allows Player 1 to delay for however long he pleases, in steps of 1 time unit.
However, if there had been an outgoing universal transition from q_2, as seen in Figure 3.3, it should be noted that Player 2 can force Player 1 to delay until breaking the bounds of the game. After one delay transition, Player 2 will be in control again, and will force Player 1 to make a move. Since no other transitions exist other than delaying, the game will be lost.
To solve this, we could put an invariant on q_2.

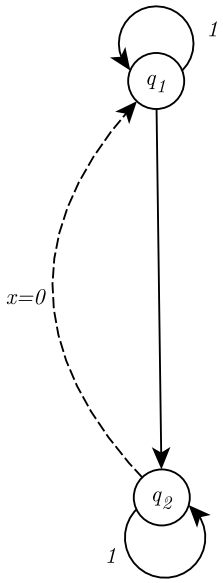
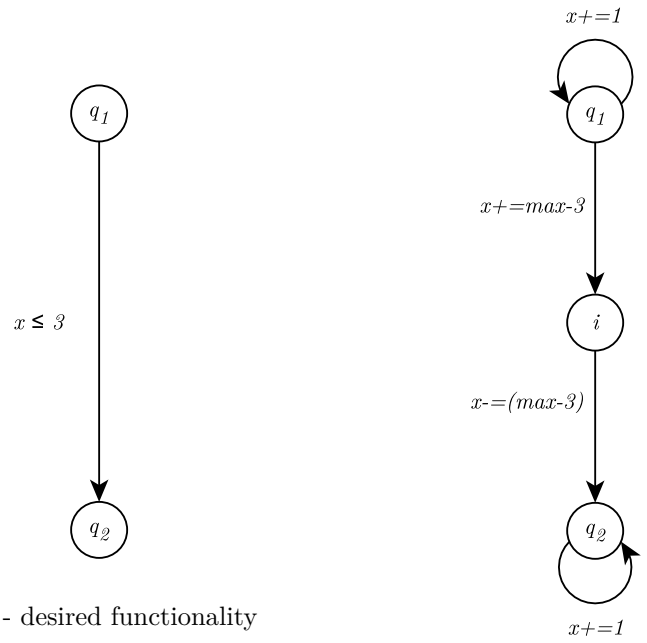


Figure 3.3: The environment can force the controller to delay indefinitely.

3.3.2 Guards

Transitions in timed games are only enabled if the current value of clocks satisfies the corresponding guards. In Figure 3.4a the transition is only possible if the clock has a value less than 3.
To simulate this, in Figure 3.4b we add the intermediate configuration i, and create the transitions $q_1 \xrightarrow{\texttt{x+=max-3}} i$, $i \xrightarrow{\texttt{x-=max-3}} q_2$.
Note that the keyword `max`, is LEG syntax referring to the upper bound of the resource on the left-hand side of the weight update, in this case `x`. We can for example assume an upper bound of 5, or expressed in LEG `weightset x:X[0,5] init 0;`. Having this upper bound on a resource which is effectively simulating a clock, means that we assume every configuration to satisfy the invariant $x \leq 5$.
Since the existential player is guaranteed to only consider winning moves in a synthesized strategy, this transition is effectively only enabled when the guard is satisfied. For example, if the upper bound is 5, $x+ = (5 - 3)$, is not winning for 4 or 5, and this holds for any value of



(a) Guard - desired functionality

(b) Guard simulated in LEG

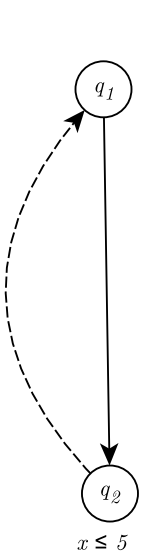
Figure 3.4: Simulating guards in LEG

upper bounds.
In the intermediate configuration i, it is important that time cannot pass, therefore, we do not add any delay transitions to this.
When leaving i, we reset the clock to the value before the 'guard' transition, or the value would be changed when checking the guard.
This effectively simulates the behaviour of a guard in an LEG Energy Game.

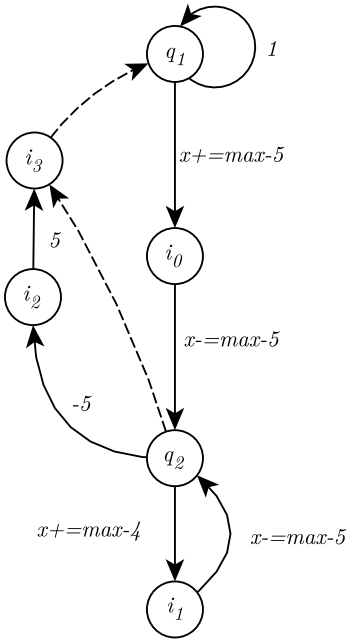
3.3.3 Invariants

Invariants are assigned to each configuration. Invariants effectively block any player from staying in a certain configuration when the conditions are no longer met. Therefore it should be noted that the universal player should not be able to occupy q_2 either if the invariant is not satisfied.

In Figure 3.5a we see the modified version of the previous example shown in Figure 3.2a. Here q_2 has an outgoing universal transition, and the invariant $x \leq 5$, where x is our single clock.
In Figure 3.5b, we see that since q_1 has no outgoing universal transitions, we simply add the looping +1 transition. The situation in q_2 is entirely different.
First, a guard is added to the transition $q_1 \longrightarrow q_2$. This guard ensures that q_2 is not reachable for values which do not satisfy the invariant. When q_2 is reached, we must only be able to delay if $x \leq 4$, since delaying by 1 if the clock is 5 would break the invariant - and leave us in q_2 with a clock value of 6. To check whether or not $x \leq 4$, we create the transition



(a) An invariant is placed in q_2.



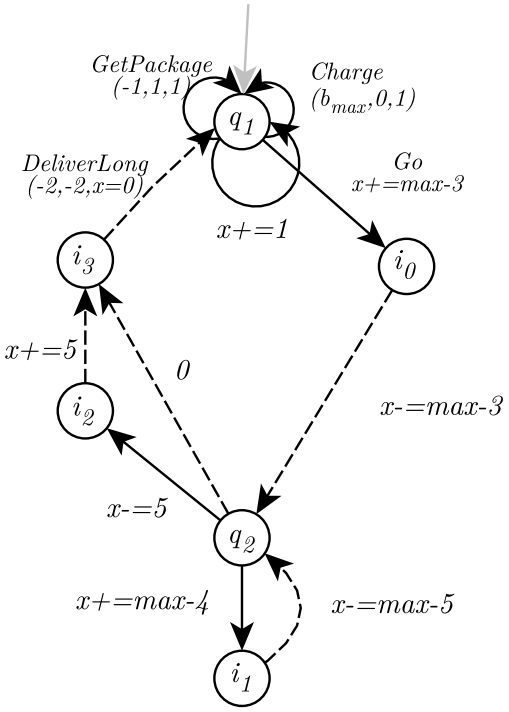
(b) Invariant simulated in LEG.

Figure 3.5: Simulating invariants in LEG.

$q_2 \xrightarrow{max-4} i_0$, to a new intermediate configuration. In this intermediate configuration, time cannot pass - the purpose of this configuration is only to check if $x \leq 4$. If x had the value 5, and the upper bound 6 for example, taking this transition would be a losing move.
When leaving the intermediate configuration, it should be noted that the value of the clock has been changed to some value, and we must return it to the value before entering i_0. This is done by subtracting the value which was added. However, in this case we wish to delay, and as such we subtract by one less than we added. We have then effectively only enabled the delay transition as long as it satisfies the invariant.
Since the universal player can take the transition from q_1 to q_2 at any time, or never, we need to give the existential player the choice to leave q_2 when the invariant is reached. This effectively simulates the behaviour of the universal player being forced to leave q_2.
In this case, we create an existential transition which subtracts by 5. Assuming that the lower bound is 0, since it is not possible for q_2 to be winning with the clock value 6, this transition is enabled if, and only if the value of x is 5.
When this transition is taken, we reach another intermediate configuration, which exists to enable another transition to reset the clock back to its original value.

3.3.4 Timed Example

Using all of the techniques just shown, we will now show how to create the timed drone seen in Figure 2.4, of Section 2.5. In the timed drone example in Figure 3.6, the `Charge` and `GetPackage` actions, consume 1 time unit. In q_1 we have added the cost of this to the weight



(a) The LEG version of the timed drone example seen in Figure 2.4.

```
1  stateset d:Location={q1,q2,i0,i1,i2,i3} init q1;
2
3  weightset b:Battery[0,4] init 0;
4  weightset p:Packages[0,2] init 0;
5  weightset x:Clock[0,6] init 0;
6
7  erules:
8  Delay: d in {q1} -> d={q1}, x+=1; //Time can pass
9  Charge: d in {q1} -> d={q1}, b += max, x+=1;
10 GetPck: d in {q1} -> d={q1}, b -=1, p+=1, x+=1;
11 Go: d in {q1} -> d={i0}, x+=max-3; //Will lose if clock guard fails
12 d in {i0} -> d={q2}, x-=max-3; //Restore clock to previous value
13 Delay: d in {q2} -> d={i1}, x+=max-4; //Check if time is less than 5
14 d in {i1} -> d={q2}, x-=max-5; //If it is, delay one
15 Delay: d in {q2} -> d={i2}, x-=5; //Invariant reached
16
17 urules:
18 d in {q2} -> d={i3}; //
19 d in {i2} -> d={i3}, x+=5; //Set clock to value before invariant
20 DelLong: d in {i3} -> d={q1}, b-=2, p-=2,x=0; //Resets clock
```

(b) The LEG code for the timed drone example.

Figure 3.6: Timed Drone in LEG.

vectors of `GetPackage` and `Charge`, and we have added the delay transition.
From q_1 to q_2, we have made the guard construction, which was discussed earlier in this section. In q_2 we have made the necessary constructions to achieve an invariant, which was

also discussed earlier in this section.
Inputting the code seen in Figure 3.6b, to EgGS, we are presented with the following strategy for q_1: In this particular example of a strategy, we are in some weighted configurations

```
1  (q1,0,0,0) -> Charge
2  (q1,0,1,0) -> Charge,Delay
3  (q1,0,1,1) -> Charge
4  (q1,0,2,0) -> Charge,Delay
5  (q1,0,2,1) -> Charge,Delay
6  (q1,0,2,2) -> Charge
7  (q1,1,0,0) -> GetPck
8  (q1,1,1,0) -> Delay,GetPck
9  (q1,1,1,1) -> GetPck
10 (q1,2,0,0) -> GetPck
11 (q1,2,2,0) -> Delay,Go
12 (q1,2,2,1) -> Delay,Go
13 (q1,2,2,2) -> Delay,Go
14 (q1,2,2,3) -> Go
15 (q1,3,1,0) -> Delay,GetPck
16 (q1,3,1,1) -> Delay,GetPck
17 (q1,3,1,2) -> GetPck
18 (q1,3,2,0) -> Delay,Go
19 (q1,3,2,1) -> Delay,Go
20 (q1,3,2,2) -> Delay,Go
21 (q1,3,2,3) -> Go
22 (q1,4,0,0) -> Delay,GetPck
23 (q1,4,0,1) -> GetPck
24 (q1,4,1,0) -> Delay,GetPck
25 (q1,4,1,1) -> Delay,GetPck
26 (q1,4,1,2) -> GetPck
27 (q1,4,2,0) -> Delay,Go
28 (q1,4,2,1) -> Delay,Go
29 (q1,4,2,2) -> Delay,Go
30 (q1,4,2,3) -> Go
```

Listing 3.7: A winning strategy for the simulated timed drone.

presented with two choices of action. Any choice in this strategy is a winning choice.
It should however be noted, that assuming a starting weighted configuration of $(q_1, \overline{0})$, some of these transitions are not reachable. It is for example not possible to be in the weighted configuration $(q1, 1, 0, 0)$, as any action before that would have added to the third resource (the clock).
This however is not an issue - it merely means that the strategy is valid for several starting weighted configurations. As such we have simulated a timed weighted energy game in LEG, and synthesized a strategy for it.

CHAPTER 4

SYMBOLIC REPRESENTATION

In this chapter we will present an approach to provide efficient implementation of the fixed point algorithm. Utilizing the expressive power of the LEG language, we present a symbolic representation of LEG energy games and show how to synthesize a strategy with this representation. The symbolic representation calls for different data structures and a somewhat different algorithmic approach, compared to an explicit implementation.

4.1 Binary Decision Diagrams

In order to achieve a symbolic representation, energy games will be encoded as quantified boolean expressions. Quantified boolean expressions may efficiently be implemented by a Reduced Ordered Binary Decision Diagram (ROBDD). In this section we will give a short description of BDDs and define what it means for a BDD to be Reduced and Ordered. BDDs were introduced in [7]. The definitions in this section are based on [7] and [4].

Definition 4.1 *Binary Decision Diagram*
A binary decision diagram is a rooted, directed acyclic graph with a vertex set V. Every vertex $v \in V$ is either terminal or non-terminal.

- *The two terminal vertices are labelled* 0 *and* 1.
- *A non-terminal vertex corresponds to a test on a boolean variable. As such each vertex is associated with a variable, $var(v) = var$. The outgoing edges are defined by tests on this variable. Low ($low(v)$), corresponding to the assignment of 0, is graphically represented by a dashed line. High ($high(v)$), corresponding to the assignment of 1, is graphically represented by a solid line.*

BDDs may be used to compactly represent any boolean expression. This allows for many practical applications, and many problems can be encoded as boolean expressions. Since

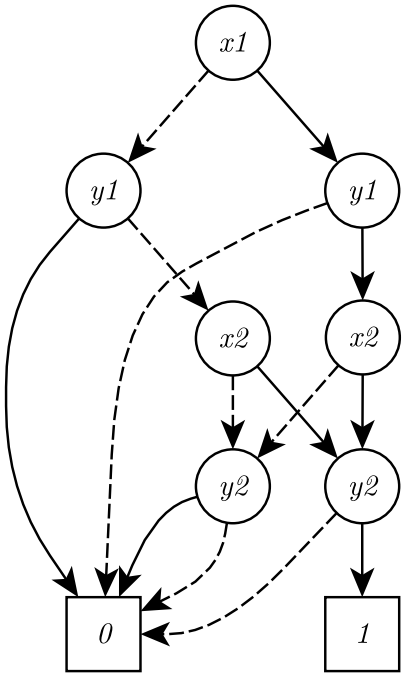


Figure 4.1: A Binary Decision Diagram.

BDDs can represent any boolean expression, they also support any boolean operation. An example of a BDD is shown in Figure 4.1.

Definition 4.2 *Ordered Binary Decision Diagram*
A Binary Decision Diagram is ordered (OBDD), if for all paths through the graph, variables follow a given linear order O, e.g. $O = x_1 < x_2 < x_3 < ... < x_n$

Note that a OBDD can be reordered at any time but must follow the ordering O through all paths in the graph. For certain BDDs, ordering of variables can be of quite some importance in regards to the size of the graph.

Definition 4.3 *Reduced Ordered Binary Decision Diagram*
An OBDD is reduced if it satisfies two conditions.

- *Every vertex in the graph must be* ***unique****. This means that no two vertices will have the same variable name, with the same low- and high-successors.*
- *Every test on a vertex must be non-redundant. This means that a given vertex must have different low- and high-successors.*

An example of an ROBDD, and a non-reduced OBDD is given in Figure 4.2, along with an ROBDD without the 0-terminal node (the standard representation).

For the rest of the report, when we refer to BDDs, we are always referring to ROBDDs.

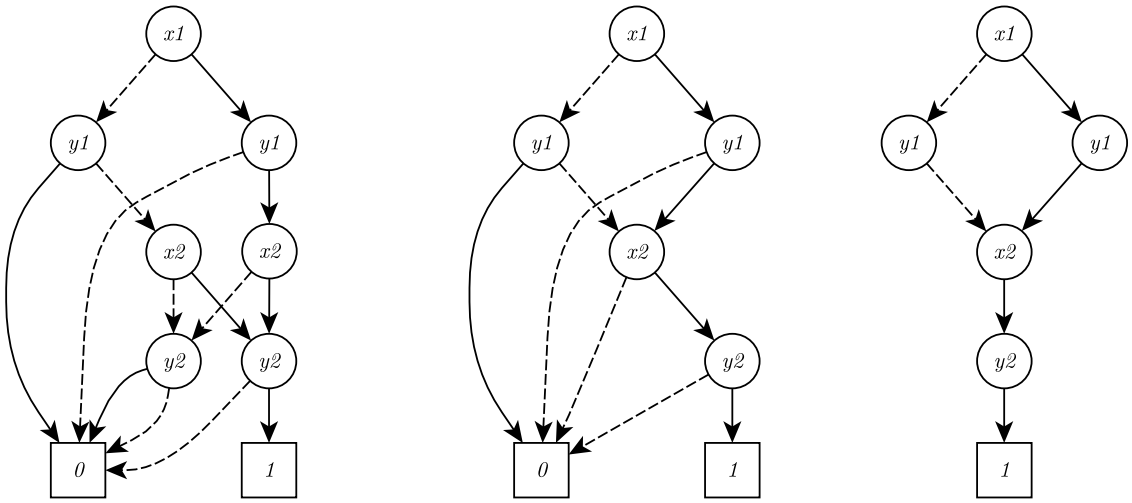


Figure 4.2: Far left: OBBD. Middle: ROBDD. Far Right: ROBDD with negative terminal removed.

4.1.1 BDD Operations

The BDD data structure is very compact and also efficient for handling quantified boolean expressions. In addition, the ROBDD data structure has some convenient properties. These properties are based on the canonicity lemma, given in [4]. The canonicity lemma states that for any boolean expression there is one, and only one, ROBDD over the same variable ordering. A consequence of this is the ability to check whether a quantified boolean expression is a tautology or satisfiable in constant time.
BDDs support all operators which are also supported by boolean expressions, as well as the quantifiers $\exists$ and $\forall$.
We will not go into detail on the operations of BDDs, but we will comment briefly on some of them. When working with BDDs, the most used operations are Apply and Restrict.
By $\mid v \mid$ we denote the number of vertices in a BDD. By v, we denote the set of vertices and edges making up a BDD.
Any operator used in boolean expressions can be expressed in a call to $\textsc{Apply}(v_1, v_2, op)$ where op is the boolean operator, and v_1 and v_2 are BDDs on which to perform the operation. It works by going through each vertex of the BDDs from the root, and construction a new BDD with the applied operator.
Restrict is called as $\textsc{Restrict}(v, x, b)$, where v is a BDD, x is a set of variables, and b is a truth assignment. It works by going through each vertex, and if $var(v) = x$, the vertex's high or low branch is removed, depending on the truth assignment b. For example, if b is *true*, the low edge is removed. This restriction returns a BDD for which only the given truth assignments will satisfy the given vertices. The complexities of these two algorithms are shown in Table 4.1. The quantifiers $\exists$ and $\forall$ are defined as follows:

$$\exists x.t = t[0/x] \vee t[1/x]$$
$$\forall x.t = t[0/x] \wedge t[1/x]$$

Algorithm	Complexity
$\textsc{Apply}(v_1, v_2, op)$	$O(\mid v_l \mid \cdot \mid v_r \mid)$
$\textsc{Restrict}(v, w, b)$	$O(\mid v \mid)$

Table 4.1: Complexities of Apply and Restrict

Where x is a set of variables to satisfy the truth assignment t.
As such, both quantifiers can be implemented by two calls to Restrict, and one call to Apply.
It is possible to determine that two boolean functions are the same, by constructing their ROBDDs ϕ_1 and ϕ_2 and verifying that $\phi_1 \Leftrightarrow \phi_2$ is a tautology.

4.2 Quantified Boolean Encoding

In this section we will present the encoding of LEG Energy Games as quantified boolean expressions. Encoded as quantified boolean expressions, we will show how symbolic representation is achieved.

As previously shown, LEG is an expressive language for expressing energy games. In order to utilize the expressiveness of LEG during the computation, the applied data structures must also be very expressive. The BDD data structure is a data structure which can very compactly represent quantified boolean expressions. This requires a symbolic encoding of the game. This symbolic representation must be able to represent all possible weighted configurations.

In order to explain the encoding of LEG into quantified boolean expressions, the example in Figure 4.3 is used. Figure 4.3 shows the Energy Game resulting from the LEG-code in Listing 4.1. First we detail how to encode the statesets and weightsets into boolean variables and values. Next we show how rules are encoded using boolean expressions. Using these two transformation techniques, we show how to compute a fixed point using boolean expressions and quantifications.

4.2.1 Configuration Encoding

To achieve configurations encoded using boolean variables, each configuration must correspond to a unique assignment of boolean variables. Recall that each configuration is a tuple of states, one state from each of the statesets. Each state in each stateset can be encoded by assignment of boolean variables. In the example, the states from stateset `X` would require one boolean variable to encode, namely $\texttt{x}_0$. `one` from stateset `X` would be encoded as $\texttt{x}_0 \Leftrightarrow false$ and the state `two` would be encoded as $\texttt{x}_0 \Leftrightarrow true$. We shall use the equivalent encoding $\neg\texttt{x}_0$ for $\texttt{x}_0 \Leftrightarrow false$ and $\texttt{x}_0$ for $\texttt{x}_0 \Leftrightarrow true$ for the remainder of the chapter. The boolean variables required to encode a stateset is equivalent to the bits required to encode the amount of states present in the stateset. Thus if a stateset had 100 states, it would require 7 boolean variables, as 7 bits are required to express the value 99. As both statesets in our example have 2 states,

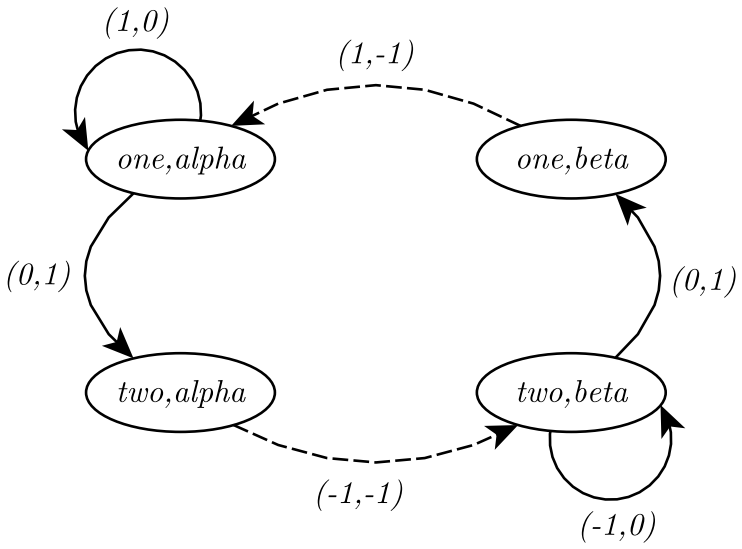


Figure 4.3: Example game.

```
1  stateset x:X={one, two} init one;
2  stateset y:Y={alpha, beta} init alpha;
3
4  weightset a:A[0,4] init 0;
5  weightset b:B[0,1] init 0;
6
7  erules:
8  x in {one}, y in {alpha} -> a+=1;
9  x in {two}, y in {beta} -> a-=1;
10 x in {one}, y in {alpha} -> x={two}, b+=1;
11 x in {two}, y in {beta} -> x={one}, b+=1;
12
13 urules:
14 x in {two}, y in {alpha} -> y={beta}, a-=1, b-=1;
15 x in {one}, y in {beta} -> y={alpha}, a+=1, b-=1;
```

Listing 4.1: The LEG code for the example in Figure 4.3.

they require only 1 boolean variable each.
Weightsets have a similar encoding. In the example the weightset `A` contains the values from 0 to 4. 3 variables are required to encode 5 values, thus the weightset `A` needs the three variables: `a0`, `a1`, `a2`.
Following the example, any weighted configuration can be expressed by use of the boolean variables `x0`, `y0`, `a0`, `a1`, `a2`, `b0`. Note that this also allows the encoding of weighted configurations outside the bounds given by the weightsets. We will address this problem later. Table 4.2 shows the statesets and weightsets of our example and their associated boolean encoding.

Matching the boolean variables with a weighted configuration is done by means of a boolean

Stateset	Boolean encoding	Boolean prime variables
`stateset x:X=one, two init one;`	x_0	x'_0
`stateset y:Y=alpha, beta init alpha;`	y_0	y'_0
Weightsets	Boolean encoding	Boolean prime variables
`weightset a:A[0,4] init 0;`	$w_0w_1w_2$	$w'_0w'_1w'_2$
`weightset b:B[0,1] init 0;`	v_0	v'_0

Table 4.2: Statesets and weightsets with their associated boolean encoding.

expression, specifically the conjunction of all the variables. For example, given the weighted configuration `(x,y,a,b)=(one, beta, 3, 1)`, the corresponding boolean expression would be $\neg x_0 \wedge y_0 \wedge a_0 \wedge a_1 \wedge \neg a_2 \wedge b_0$. To simplify the following explanations, we introduce the following shorthand notation.
$\overline{s}$ is a vector representing the boolean variables $s_0, \ldots, s_n$, either a stateset or a weightset.

4.2.2 Rule Encoding

A rule in LEG can give rise to transitions from multiple configurations to multiple configurations, depending on the states the rule is enabled on. In this example we consider a simple transition from a single configuration to another configuration (`(one,alpha)` $\longrightarrow$ `(two,alpha)`). This transition is modelled by the existential rule in Line 10 in Listing 4.1. The rule is enabled only when `x=one` and `y=alpha`. To model the transition as a boolean expression, we must consider the weighted configuration before and after the transition. To model the resulting weighted configuration, we need the same variables required to encode the enabling weighted configuration, but the variables must be distinguishable from each other. We introduce a prime variable for each boolean variable, as shown in Table 4.2. The prime variables will always relate to a resulting weighted configuration and thus only be present in boolean expressions for rules.

Consider the rule

$$\texttt{x in }\{\texttt{one}\},\ \texttt{y in }\{\texttt{alpha}\} -> \texttt{x} = \{\texttt{two}\}, \texttt{b+} = 1;. \tag{4.1}$$

This rule states that a transition from `(one,alpha)` to `(two,alpha)` is possible, while adding 1 to the weight b. To model this, let us first consider the states. The configuration that enables this rule must have $\neg x_0 \wedge \neg y_0$ and it must result in a configuration $x_0 \wedge \neg y_0$. In addition the rule must add 1 to the weight `b`, which we model with the abstract predicate $add(\overline{b}, \overline{b'}, +1)$. Any weight or stateset that is not mentioned must remain the same before and after the rule. Thus the boolean encoding of the rule is

$$\neg x_0 \wedge \neg y_0 \wedge x'_0 \wedge \neg y'_0 \wedge add(\overline{a}, \overline{a'}, 0) \wedge add(\overline{b}, \overline{b'}, +1). \tag{4.2}$$

Table 4.3 shows the encoding for every rule in the example.

In case the rule allows a non-deterministic choice between enabling or resulting states, this is encoded as the disjunction of each possible state. For example if the rule had been

$$\texttt{x in }\{\texttt{one}\},\ \texttt{y in }\{\texttt{alpha}\} -> \texttt{x} = \{\texttt{two}\}, \texttt{y} = \{\texttt{alpha}, \texttt{beta}\}\texttt{b+} = 1;, \tag{4.3}$$

Existential rules	Boolean encoding
`x in {one}, y in {alpha} -> a+=1;`	$\neg \mathtt{x}_0 \wedge \neg \mathtt{y}_0 \wedge$ $\neg \mathtt{x}'_0 \wedge \neg \mathtt{y}'_0 \wedge add(\overline{\mathtt{a}}, \overline{\mathtt{a}'}, +1) \wedge add(\overline{\mathtt{b}}, \overline{\mathtt{b}'}, 0)$
`x in {two}, y in {beta} -> a-=1;`	$\mathtt{x}_0 \wedge \mathtt{y}_0 \wedge$ $\mathtt{x}'_0 \wedge \mathtt{y}'_0 \wedge add(\overline{\mathtt{a}}, \overline{\mathtt{a}'}, -1) \wedge add(\overline{\mathtt{b}}, \overline{\mathtt{b}'}, 0)$
`x in {one}, y in {alpha} -> x={two}, b+=1;`	$\neg \mathtt{x}_0 \wedge \neg \mathtt{y}_0 \wedge$ $\mathtt{x}'_0 \wedge \neg \mathtt{y}'_0 \wedge add(\overline{\mathtt{a}}, \overline{\mathtt{a}'}, 0) \wedge add(\overline{\mathtt{b}}, \overline{\mathtt{b}'}, +1)$
`x in {two}, y in {beta} -> x={one}, b+=1;`	$\mathtt{x}_0 \wedge \mathtt{y}_0 \wedge$ $\neg \mathtt{x}'_0 \wedge \mathtt{y}'_0 \wedge add(\overline{\mathtt{a}}, \overline{\mathtt{a}'}, 0) \wedge add(\overline{\mathtt{b}}, \overline{\mathtt{b}'}, +1)$
Universal rules	Boolean encoding
`x in {two}, y in {alpha} -> y={beta}, a-=1, b-=1;`	$\mathtt{x}_0 \wedge \neg \mathtt{y}_0 \wedge$ $\mathtt{x}'_0 \wedge \mathtt{y}'_0 \wedge add(\overline{\mathtt{a}}, \overline{\mathtt{a}'}, -1) \wedge add(\overline{\mathtt{b}}, \overline{\mathtt{b}'}, -1)$
`x in {one}, y in {beta} -> y={alpha}, a+=1, b-=1;`	$\neg \mathtt{x}_0 \wedge \mathtt{y}_0 \wedge$ $\neg \mathtt{x}'_0 \wedge \neg \mathtt{y}'_0 \wedge add(\overline{\mathtt{a}}, \overline{\mathtt{a}'}, +1) \wedge add(\overline{\mathtt{b}}, \overline{\mathtt{b}'}, -1)$

Table 4.3: Rules and their associated boolean encoding.

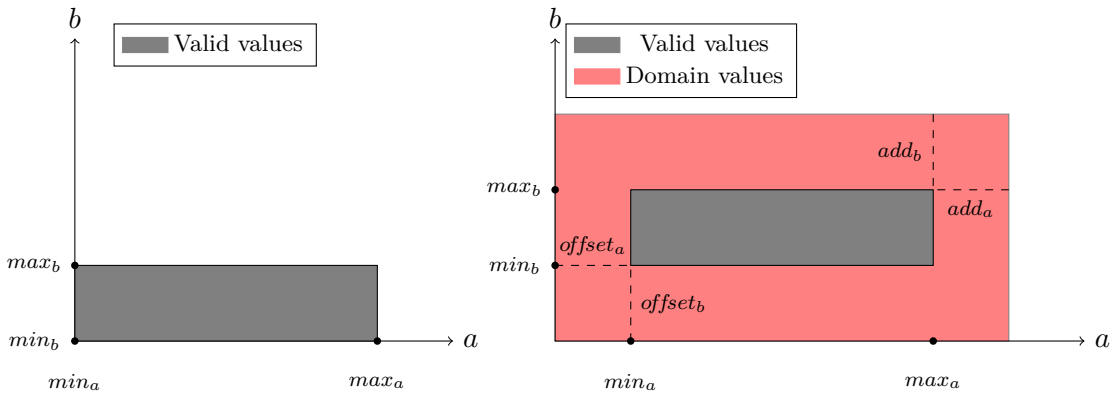
i.e. `y` could result in either `alpha` or `beta`, the encoding would be

$$\neg \mathtt{x}_0 \wedge \neg \mathtt{y}_0 \wedge \mathtt{x}'_0 \wedge (\neg \mathtt{y}'_0 \vee \mathtt{y}'_0) \wedge add(\overline{\mathtt{a}}, \overline{\mathtt{a}'}, 0) \wedge add(\overline{\mathtt{b}}, \overline{\mathtt{b}'}, +1). \tag{4.4}$$

There are several ways to represent the predicate $add(\overline{\mathtt{w}}, \overline{\mathtt{w}'}, k)$, where $k \in \mathbb{N}$ as a quantified boolean expression. Intuitively, the predicate will encode a boolean expression describing each value that can be expressed through the the variables $\overline{\mathtt{w}}$. This is then associated with the boolean encoding of the value resulting from applying the arithmetic described in k to the value, using the variables $\overline{\mathtt{w}'}$. That is, for each value $i \in W$, where $W = [0, m]$, we encode the value i as b^i using the variables $\overline{\mathtt{w}} = b^i$. We proceed to encode $i + k$ as b^{i+k} using the variables $\overline{\mathtt{w}} = b^{i+k}$. We then related the encodings as a conjunction, i.e. $(\overline{\mathtt{w}} = b^i \wedge \overline{\mathtt{w}'} = b^{i+k})$. As this is done for each value $i \in W$, the disjunction of all the encodings is needed, i.e.

$$(\overline{\mathtt{w}} = b^0 \wedge \overline{\mathtt{w}'} = b^{0+k}) \vee \cdots \vee (\overline{\mathtt{w}} = b^m \wedge \overline{\mathtt{w}'} = b^{m+k}) \tag{4.5}$$

Recall that the boolean encoding is able to express invalid values, e.g. the variables $\mathtt{a}_0\mathtt{a}_1\mathtt{a}_2$ could express the value 7 through $\mathtt{a}_0 \wedge \mathtt{a}_1 \wedge \mathtt{a}_2$ even though the weightset variable `a` is only allowed to assume values up to 4. By encoding *add* for only the valid values, we have ensured that only valid values will count as enabling. The rules may still result in weights outside the specified bounds, as they should.
There is a small issue with resulting weights outside the specified bounds. These values may be less than 0 or assume a higher value than the boolean variables are able to express. For this reason, and only during computation, the values are increased by an offset corresponding to the greatest subtraction specified by any rule. Similarly, when the boolean variables are created, enough variables are created to accommodate the addition of the offset, the maximum value of the weight and the greatest value any rule adds to the weight. An illustration of this is given in Figure 4.4, with the original values specified by the LEG syntax illustrated in Figure 4.4a and the resulting domain values shown in Figure 4.4b.



(a) Valid values. (b) Domain values with offset.

Figure 4.4: Valid values and domain values before and after adding offset and maximal addition.

During computations all values are subject to an offset. $offset_w$ is the greatest value subtracted from the weight w. add_w is the greatest value added to the weight w. During the computation, the greatest value that can be assumed by w are given by $domain_w = max_w + offset_w + add_w$.

4.3 Symbolic Fixed Point Algorithm

After constructing every rule, we are ready to apply those rules in computation. In this section we will show an adaptation of the previously shown fixed point algorithm in Algorithm 1. The implementation of the fixed point algorithm on the symbolic encoding is implemented as constructions of BDDs.
We denote by $E(x, x')$ the disjunction of existential rules over x and x', and by $U(x, x')$ the disjunction of universal rules over x and x'. The predicate W_0 is initially true for any valid state or weight. This is achieved by encoding the valid weighted configurations, as by the example on weight a:

$$V_a = (\neg \mathtt{a}_0 \wedge \neg \mathtt{a}_1 \wedge \neg \mathtt{a}_2) \vee (\mathtt{a}_0 \wedge \neg \mathtt{a}_1 \wedge \neg \mathtt{a}_2) \vee (\neg \mathtt{a}_0 \wedge \mathtt{a}_1 \wedge \neg \mathtt{a}_2) \vee (\mathtt{a}_0 \wedge \mathtt{a}_1 \wedge \neg \mathtt{a}_2) \vee (\neg \mathtt{a}_0 \wedge \neg \mathtt{a}_1 \wedge \mathtt{a}_2) \quad (4.6)$$

W_0 is then initially defined by

$$W_0 = V_{ss_1} \wedge \cdots \wedge V_{ss_n} \wedge V_{ws_1} \wedge \cdots \wedge V_{ws_m}, \quad (4.7)$$

where ss_i is a stateset and ws_j is a weightset. Thus the fixed point step BDD construction is given by:

$$W_{n+1}(x) = W_n(x) \wedge (\exists x'.[E(x, x')] \implies \exists x'.[E(x, x') \wedge W_n(x')]) \wedge \forall x'.[U(x, x') \implies W_n(x')] \quad (4.8)$$

Where W_n is the n'th assumed solution space. This means that the symbolic fixed point algorithm is given by Algorithm 3. When we reach a fixed point, we have reached the BDD

Algorithm 3 Symbolic fixed point algorithm for LEG games.

while $(W_{n+1}(x)$!= $W_n(x))$ **do**
$W_{n+1}(x) = W_n(x) \wedge (\exists x'.[E(x,x')] \implies \exists x'.[E(x,x') \wedge W_n(x')]) \wedge \forall x'.[U(x,x') \implies W_n(x')]$

which represents all possible weighted configurations which are winning. By checking against this BDD it is easy to verify if a given weighted configuration is part of the winning space. This is, for instance, good to check for any given initial weighted configuration.
There are several reasons why Algorithm 3 is easy to implement using BDDs. Given that every part of the algorithm, save the iteration, is quantified boolean expressions, it is possible to utilise the efficient representation provided by BDDs. In addition, BDD operations are efficient. Finally, the comparison of two BDDs is a very efficient operation, as described in Section 4.1 .

CHAPTER 5

IMPLEMENTATION

In this section we elaborate on the overall architecture of the implementation. We will also explain the frameworks used, such as BuDDy and ANTLR4, and how they interact with the architecture.
The tool is currently implemented in C♯.NET 4.0. As such it is built targeting the Windows platform. However, the implementation could easily be ported to other platforms through another C♯ implementation such as the Mono Project [15].
The current GUI implementation is done in WPF (Windows Presentation Foundation)[2], which is platform-specific and as such cannot be ported to other platforms. However, the tool can function as a command-line client as well on other platforms, or have separate GUI implementation depending on platforms. Using WPF however, allows for hardware acceleration when rendering. WPF uses Direct3D for rendering, which allows the rendering process to be done via the GPU. Having hardware acceleration may aid when viewing very large graphs and diagrams.

5.1 Architecture

The overall architecture is shown in the diagram in Figure 5.1. Shown in this diagram is the various components which make up EgGS. It should be noted that this is not a complete component diagram, as some components, such as GUI and other less relevant components are not shown. There are three main components which make up the EgGS system - the LEG compiler, the BuDDy C♯ interface, and LEGProgram (the core system).
ANTLR4 is a third-party tool which assists in generating the lexer and parser classes for the LEG compiler. ANTLR4 is discussed in detail in Section 5.3. The lexer and parser is then inserted into the LEG compiler code. From here a listener class is called by observer pattern which uses the loader to initialize the GameBuilder with some LEG code.
Depending on the type of game selected the core component LEGProgram will either explicitly construct a game by listing every possible weighted configuration of the game, and running a fixed point algorithm on that assumed winning space.

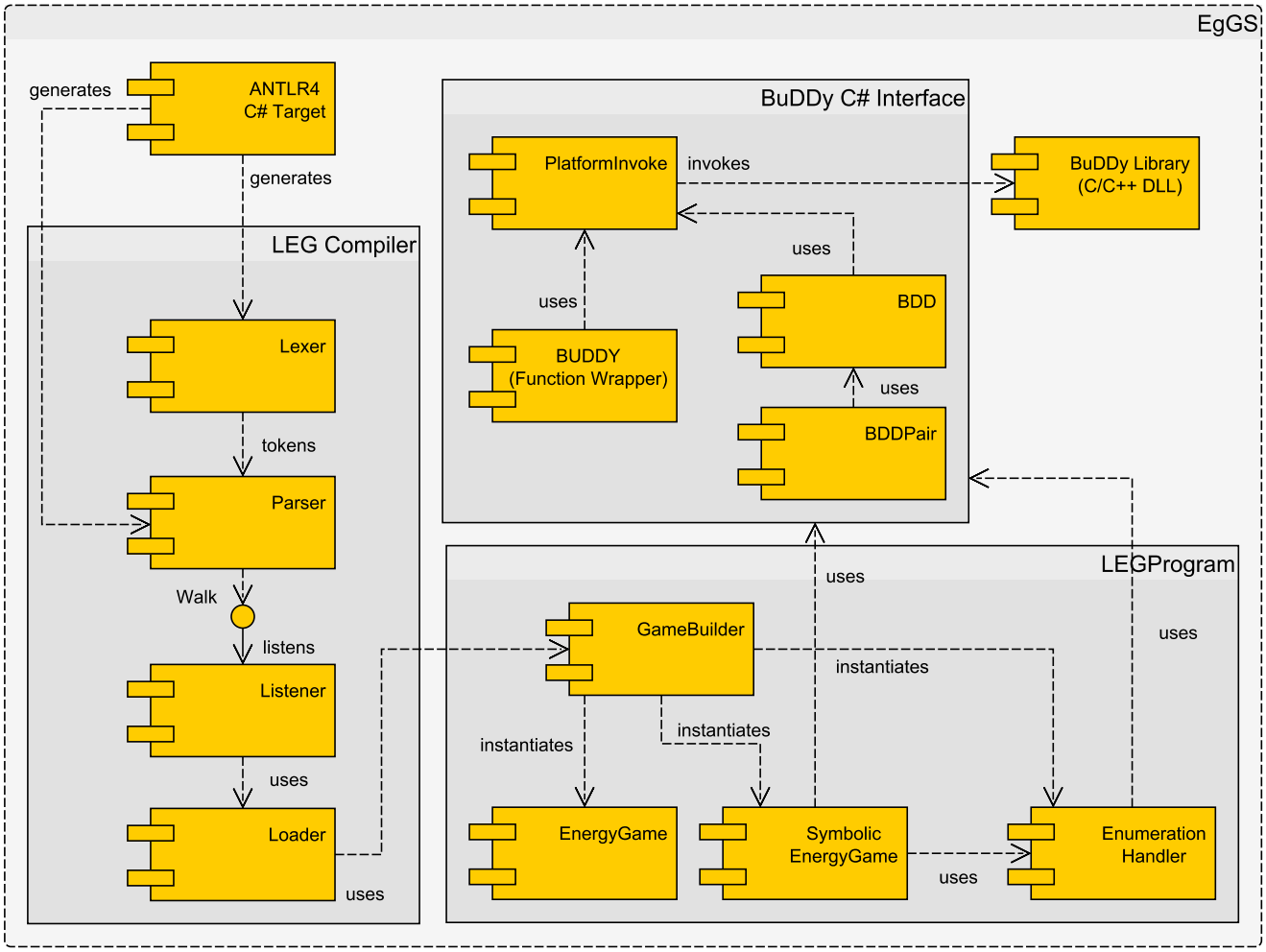


Figure 5.1: The overall architecture of the tool.

If a symbolic game is selected, the core depends on the BuDDy C♯ interface. BuDDy is an efficient Binary Decision Diagram library with highly efficient BDD operations and data structures. BuDDy is discussed in detail in Section 5.2. Using BuDDy, the core constructs a set of BDDs which symbolically encodes the entire LEG Energy Game. Then, a symbolic fixed point algorithm is computed on this set of rules to achieve a winning space. This winning space can then be presented as a list of all winning weighted configurations, or as a BDD describing all the winning weighted configurations.

5.2 BuDDy and the BuDDy C♯ interface

BuDDy [14] is a C/C++ Binary Decision Diagram library. It has highly efficient vectorized BDD operations, automated garbage collection, and many features for working with BDDs. Included in the library is a C++ interface.
In the implementation of EgGS, a custom C♯ interface was created. Using BuDDy as a C

library, we offload the main computational work to a more low-level implementation, while retaining the strong features of C♯ for the rest of the EgGS implementation.
To achieve this, BuDDy was compiled as a class library, which is then imported into the EgGS source code. In this section we will elaborate on this C♯ interface to BuDDy.
If we refer to the main architecture in Figure 5.1, it is apparent that the only class to call the BuDDy library directly is `PlatformInvoke`. Any other class in the interface will have to use any of the imported methods in it. Creating BDDs in BuDDy is done by declaring boolean variables. Each BDD is then constructed by combining them with some operator in a call to `bdd_apply`. This method takes as input pointers to two BDDs, and an integer representing which operator to apply.

5.2.1 Calling a BuDDy function

The `BUDDY` class serves as a wrapper for any function needed to manipulate BDDs. A function such as `bdd_apply(BDD l, BDD r)` is called by calling the static method `BUDDY.Apply(BDD bdd1, BDD bdd2, BUDDY.Operator op)`. Each of these wrapper functions then calls the function on the pointer of the BDD. This wrapper class also handles any information that is relevant when calling BuDDy functions, *e.g.* an enumeration representing each operation. In

```
1  public class BUDDY
2  {
3  #region Defines
4    /// <summary>
5    /// The list of operators supported by bdd_apply
6    /// </summary>
7    public enum Operator
8    {
9        /* ... */
10     }
11 #endregion
12 #region Kernel.c
13     /// <summary>
14     /// Initializes the BuDDy package
15     /// </summary>
16     public static void Init(int number_of_nodes, int cachesize)
17     {
18       bddVars.Clear();
19         initialized = true;
20         Console.WriteLine("Buddy init");
21         PlatformInvoke.bdd_init(number_of_nodes, cachesize);
22     }
23 }
```

Listing 5.1: Excerpt of the BUDDY wrapper class.

Listing 5.1 we see an excerpt of the `BUDDY` class. An example of a wrapped function is the static method `Init`.
When using BuDDy the library must first be initialized. By calling `BUDDY.Init`, and passing the initial number of nodes to be allocated, and some choice of cache size. This cache size determines how often a memory allocation takes place. If 100 BDD nodes are cached, 100 more will be allocated every time that amount is reached. This can affect performance, and

as such when initializing BuDDy the size of the energy game in question must be taken into account.
`BUDDY` calls the imported functions in `PlatformInvoke`. An example of such an imported function is shown in Listing 5.2. The signature of most of these methods are pointers or a

```
1 [DllImport(@"BuDDyHelper.dll", CallingConvention = CallingConvention.Cdecl)]
2 public static extern IntPtr bdd_apply(IntPtr bdd, IntPtr bdd2, int operator_choice);
```

Listing 5.2: `bdd_apply` imported into `PlatformInvoke`.

primitive data type. This allows inter-communication between managed and unmanaged code. Every method in BuDDy is imported in this manner, and then wrapped in the `BUDDY` class. When calling a method in `BUDDY`, the pointer is not given, but an object representing a BDD is.

5.2.2 Representing BDDs

To create a BDD or a BDD variable, the class `BDD` represents each BDD as an object, as seen in Listing 5.3, which is an excerpt of that class. Note that this does not mean that every node in BuDDy's table is represented as an object, only the pointer to that BDD is. When constructing a new BDD variable, we generally want to give it an index in the ordering, and a name.
BDDs are represented with a platform-specific pointer to the node in the BuDDy library, a nullable integer referring to the index if the BDD is a variable, and in that case, the name of the variable. These parameters are passed to the constructor of `BDD`, which then calls `IthVar`, which calls `bdd_ithvar` in BuDDy with the given index. A minor loss of efficiency with this approach is that some BDDs are only freed in BuDDy's memory, when the C♯ garbage collector frees that BDD object.
The memory management in BuDDy is implemented as a custom garbage collector. This garbage collector depends on reference counting. When a variable is created, it automatically calls `bdd_addref`, which increases the reference count of that node by 1. A call to `bdd_ithvar` also calls `bdd_addref`.
When the C♯ garbage collector frees a BDD object, the destructor decreases the reference count of that node by one.

The names of each BDD variable is saved in the wrapping class `BUDDY`. Here a list is held of all the names of the declared variables. This class also holds all relevant methods for constructing the rules that can be expressed in LEG, and automatization of declaring the required number of variables for representing a stateset or weightset.

In Listing 5.4, we see the implementation of the static method `MakeAddition`. Given the necessary information, such as the upper bound and lower bound, and the number to add by, this method returns a BDD representing that addition. Note that the prime variables in this BDD may in fact go outside the bounds of the game. This detail is important to the fixed point algorithm, as discussed in Section 4.2.2.
Note that the implementation of `MakeSubtraction` is identical, with the exception of sub-

```
public class BDD
{
  public IntPtr pointer;
    public int? i;
    public string name;

    #region Constructors and related functions
        /* ... */
    /// <summary>
    /// Construct a new named BDD Variable.
    /// Adds a name and a reference to the ith variable.
    /// </summary>
    /// <param name="_i">The variable index.</param>
    /// <param name="_name">The variable name.</param>
    public BDD(int _i, string _name)
    {
        i = _i;
        pointer = IthVar(_i);
        name = _name;
        lastIndex = _i;
        BUDDY.bddVars.Add(this);
    }
          /* .. */

    /// <summary>
    /// Creates the ith variable in BuDDy.
    /// </summary>
    private IntPtr IthVar(int i)
    {
        return PlatformInvoke.bdd_ithvar(i);
    }
    /// <summary>
    /// Destructor - call DelRef on all destroyed objects in C# code
    /// </summary>
    ~BDD()
    {
        if(BUDDY.initialized)
           this.DelRef();
    }
            /* ... */
}
```

Listing 5.3: Excerpt of the BDD class.

tracting the operand in the prime variables.

To declare the number of required variables to represent a weightset, the method `BoundedVarArray`, as seen in Listing 5.5, will automatically determine the required amount of boolean variables needed to represent values op to `maxvalue`. This method is used when declaring the variables representing the weightsets.

5.2.3 Satisfying Variable Assignments

When a BDD is created, we can ask BuDDy for all variable assignments that satisfy that BDD. In the case of having computed the winning space as a BDD, this means getting the explicit binary version of the winning space. In BuDDy, this is implemented with a call to `bdd_allsat`. In the BuDDy library, this function has this signature `void bdd_allsat(BDD`

```
public static BDD MakeAddition(int operand, int xMin, int xMax, int xOffset, BDD[] x↩
    , BDD[] x_)
{
  int bitsRequired = x.Length;
    //Bitstrings given as plain strings
    string xString = ""; //bitvalue of non-prime
    string x_String = ""; //bitvalue of prime
    BDD[] subresults = new BDD[xMax-xMin+1];
    //enumerate values over the interval of valid values
    for (int n = 0; n <= xMax - xMin; n++)
    {
      //value within Valid bounds
        xString = BitFunctions.BitString(n + xOffset + xMin, bitsRequired);
        //Value resulting from applying the operand - may go outside bounds
        x_String = BitFunctions.BitString(n + xOffset + xMin + operand, bitsRequired↩
            );
        //store the subresult
        subresults[n] = MakeBDDFromBitString(xString, x) & MakeBDDFromBitString(↩
            x_String, x_);
    }
    //Now build the entire addition rule by disjunction of all the possible values ↩
        stored in subresults
    BDD result = BUDDY.BDDFalse();
    for (int i = 0; i < subresults.Length; i++)
    {
      result = result | subresults[i];
    }
    return result;
}
```

Listing 5.4: Method for generating an addition rule.

```
/// <summary>
/// Construct BDD variables with the given name from 0 to max value.
/// </summary>
/// <param name="upperbound">The maximum value the variables must be able to ↩
    represent.</param>
/// <param name="varName">The identifier for the variables.</param>
/// <returns>Array of BDD variables.</returns>
public static BDD[] BoundedVarArray(int maxvalue, string varName)
{
  string maxvalueBit = BitFunctions.BitString(maxvalue);
  //Store all our x variables
  BDD[] x = new BDD[maxvalueBit.Length];

  //Construct all our variables
  for (int i = 0; i < maxvalueBit.Length; i++)
  {
    x[i] = new BDD(varName +(1 + i));
  }
  return x;
}
```

Listing 5.5: Method for declaring the number of required variables.

`r,bddallsathandler handler)`. That is, as input it expects a BDD for which to find all satisfying variable assignments, and a function pointer to a function which handles the variable

assignment. For every satisfying variable assignment the handler will be called. This handler has the signature `void allsatPrintHandler(char* varset,int size)`. When it is called, BuDDy will pass a char array to the function, and the handler will process that information in whatever way necessary. In this array of chars, the index refers to the variable index, and a value representing the variable assignment. False is equivalent to 0, true equivalent to 1, and in case that variable can be both true or false, -1 is given.
To accommodate this functionality, the handler passed to BuDDy is implemented in the C♯ interface.

```
1 [DllImport(@"BuDDyHelper.dll", CallingConvention = CallingConvention.Cdecl)]
2 public static extern void bdd_allsat(IntPtr bdd, [MarshalAs(UnmanagedType.↩
      FunctionPtr)] AllSatHandler allsathandler);
3
4 [UnmanagedFunctionPointer(CallingConvention.Cdecl)]
5 public delegate void AllSatHandler(IntPtr varset,int size);
```

Listing 5.6: Importing a function which takes a function pointer as an argument.

A delegate in C♯, is really just a function pointer. This function pointer is marshalled as an unmanaged type, as a function pointer. The signature of this delegate, corresponds to the signature of a handler implemented in BuDDy. This means that getting all satisfying variable assignments from the C♯ interface can be achieved in the following way:

```
1 /// <summary>
2 /// Call allsat with your own handler method
3 /// Be careful when creating such a method - do not deviate on the signature, and be↩
      aware that the pointer from c is a char* (corresponds to byte[])
4 /// </summary>
5 public static void AllSat(BDD bdd, PlatformInvoke.AllSatHandler handler)
6 {
7   PlatformInvoke.bdd_allsat(bdd.pointer, handler);
8 }
```

Listing 5.7: A call to AllSat.

An implementation of a handler which prints all the satisfying variable assignments to the console can be implemented in this manner shown in Listing 5.8. The method for getting all satisfying variable assignments then simply becomes:

```
1 public static void AllSatPrint(BDD bdd)
2 {
3   //Delegate as C function pointer.
4   PlatformInvoke.AllSatHandler handler = AllSatPrinter;
5   PlatformInvoke.bdd_allsat(bdd.pointer, handler);
6 }
```

5.2.4 Using the BuDDy C♯ Interface

Consider the example shown in Section 4.2.2, and specifically the rule `x in {one}, y in {alpha} -> a+=1`. From Listing 4.1, we know that the game in question consists of the

```
static void AllSatPrinter(IntPtr varset, int size)
{
  byte[] managedArray = new byte[size];
  Marshal.Copy(varset, managedArray, 0, size);
  int[] ints = new int[size];
  for (int i = 0; i < size; i++)
  {
    if (managedArray[i] == 255)
    {
      //Assignment irrelevant
      ints[i] = -1;
    }
    if (managedArray[i] == 1)
    {
      //Assignment positive
      ints[i] = 1;
    }
    if (managedArray[i] == 0)
    {
      //Assignment negative
      ints[i] = 0;
    }
    if (ints[i] != -1)
    Console.Write(bddVars[i].name + "->" + ints[i] + " ");
  }
  Console.Write("\n ---------------------------- \n");
}
```

Listing 5.8: `AllSatPrinter` - prints all satisfying variable assignments to the console.

statesets `x:X={one, two}` and `y:Y={alpha, beta}`.
To encode these statesets, we need one boolean variable for each of these. Additionally, we require prime variables, to encode their values after a transition. Note that the indexes given

```
BDD x = new BDD(0,"x");
BDD y = new BDD(1,"y");
BDD x_ = new BDD(2,"x'");
BDD y_ = new BDD(3,"y'");
```

here are used to determine the ordering in the resulting BDDs. To encode the rule, we create an enumeration of all possible values of the variables `x` and `y`. For this case, these are merely 0 and 1, representing the first and the second state in the statesets. Next to create the necessary values, we can use the method shown in Listing 5.5. Assume the variable a in the rule, has an upper bound of 4, and a lower bound of 0.

```
BDD[] a = BDD.BoundedVarArray(4, "a"); //a variables created
BDD[] a_ = BDD.BoundedVarArray(4, "a'");//a prime variables created
```

```
BDD rule = (!x & !y) & (!x_ & !y_) & BDD.MakeAddition(1,0,4,0,a,a_);
```

Corresponding to the logic $(\neg \mathtt{x} \wedge \neg \mathtt{y}) \wedge (\neg \mathtt{x}' \wedge \neg \mathtt{y}') \wedge add(\bar{\mathtt{a}}, \bar{\mathtt{a}}', +1)$, as seen in Table 4.3. The `&` operator, is a an overloaded operator, which performs code equivalent to the following:

```
1 BUDDY.Apply(x,y,BUDDY.Operator.And);
```

As such we have effectively created the state variables, and the weight variables necessary to represent this rule. In Section 5.4 we will show how this is used to execute the symbolic fixed point algorithm.

5.3 ANTLR4

ANTLR4[16] (ANother Tool for Language Recognition) is a tool for generating parsers. It also provides a runtime library, which is able to walk the constructed parse-trees. It is a widely used tool for parser generation, and is able to generate class files for C♯. When defining the language, we started out with code examples of how we wanted the concrete syntax to appear. After a few iterations and some feedback on this, we constructed a grammar in the form of an EBNF in [8]. Further updates and adjustments have been made to the grammar of LEG (shown in Table 5.1), but it does not impact the semantics of LEG. This grammar is defined in ANTLR's grammar language, and passed as input to ANTLR4. ANTLR4 then generates the lexer `LEGLexer` and the parser `LEGParser` over the grammar, and using the ANTLR4 C♯ runtime library, walks the parse tree. While the parse tree is being walked, the construction of an energy game is handled by an observer pattern in `LEGListener` which calls `LEGLoader`. While walking the tree from the left to the right, the listener and loader are called by observation, and receives the relevant tokens and parsing contexts and passes this to methods in the builder to build an energy game. Depending on the type of game chosen (symbolic or non-symbolic), a game is instantiated by the `GameBuilder`.

cletter	::=	[**A**-**Z**]
lletter	::=	[**a**-**z**]
letter	::=	cletter | lletter
digit	::=	[**0**-**9**]
integer	::=	digit (digit)*
transitionname	::=	cletter (lletter | cletter)* **:**
operator	::==	**+** | **-** | *****
statesetdeclaration	::=	**stateset** statevariable **:** statesetidentifier **=** stateset **init** stateidentifier**;**
statevariable	::=	lletter (letter| integer)*
statesetidentifier	::=	cletter (letter | integer)*
stateset	::=	**{** stateidentifier(**,** stateidentifier)* **}**
stateidentifier	::=	(integer | lletter) (letter | integer)*
weightdeclaration	::=	**weightset** weightvariable **:** weight **;**
weightvariable	::=	lletter (letter| integer)*
weight	::=	weightidentifier **[** integer **,** integer **]** **init** integer
weightidentifier	::=	cletter (letter | integer)*

ruledeclarations	::=	(erule | urule | eruleblock | uruleblock)*
erule	::=	< rule > ;
urule	::=	[rule] ;
eruleblock	::=	**erules:** rule ; (rule ;)*
uruleblock	::=	**urules:** rule ; (rule ;)*
rule	::=	(transitionname condition - > consequence) | (condition - > consequence)
condition	::=	comparison (, comparison)*
consequence	::=	(stateassignment|weightassignment)(,(stateassignment | weightassignment))*
comparison	::=	statevariable **in** (stateset | statesetidentifier)
stateassignment	::=	statevariable = (stateset | statesetidentifier)
weightassignment	::=	weightvariable (= | operator=) (equation | **max** | **min**)
operator	::=	+ | - | *
equation	::=	((equation operator equation))| ((equation)) | (number | weightvariable | **max** | **min**) (operator (integer | weightvariable | **max** | **min**))*
model	::=	statesetdeclaration (statesetdeclaration)* weightdeclaration (weightdeclaration)* ruledeclarations

Table 5.1: EBNF for LEG .

Among the updates to the LEG grammar, we have the addition of comments (not shown in the EBNF, since this works by skipping any tokens after `//`), changes to the token structure, to fix some bugs, and the possibility to name transitions. This can be seen in `rule` in Table 5.1. Additionally, as an error, the keywords `max` and `min` were missing from `equation`, and have been added. Adjustments have been made for full paranthesis support in equations.

5.4 LEGProgram - the EgGS core

The core of EgGS is mainly composed of the game builder, and the algorithms for computing the greatest fixed point of a given game expressed using LEG syntax. When the `GameBuilder` class receives instructions from the `LEGLoader`, depending on the choice of algorithm, it either constructs a class of the type `EnergyGame`, or of the type `SymbolicEnergyGame`. After constructing an `EnergyGame`, the algorithm will examine every single weighted configuration in the assumed winning space to find a greatest fixed point. This algorithm is seen in Listing 5.9 If the instantiated class is of type `SymbolicEnergyGame`, an `EnumerationHandler` class will be

```
public LEGConfigurationSpace<T> FixedPointAlgorithm()
{
  //Declarations and initializations
    LEGConfigurationSpace<T> w = weightedConfigurationSpace; //Current configSpace
    LEGConfigurationSpace<T> w_old = new LEGConfigurationSpace<T>(); //ConfigSpace ↩
        for comparison
    Stopwatch stopWatch1 = new Stopwatch(); //iteration timer
    Stopwatch stopWatch2 = new Stopwatch(); //total timer
    iterations = 0;
    while (!w_old.Equals(w))
    {
        iterations++;

        w_old = w;
        w = FF(w_old);
        Console.WriteLine("\nIteration {0} done",iterations);
    }

    Console.WriteLine("Fixed point found in {0} iterations",iterations);
    return w;
}
```

Listing 5.9: Fixed-point algorithm, main part.

instantiated. This class holds all information related to the encoding of the variables which have been created in BuDDy. This means that for any weightset declared in LEG, when comparing the symbolic values for the variables representing that weightset, the enumerationhandler holds information about the offset, upper and lower bound and greatest addition related to the weightset. For example, given a string of bits representing the satisfying assignment to the state variables, enumerationhandler gives the name of the state that the variable assignment corresponds to, as seen in Listing 5.10.

```
public string GetState(string _bits, string _statesetVariable)
{
  int debug = BitFunctions.GetInt(_bits);
    return StatesetTable[_statesetVariable][debug];
}
```

Listing 5.10: EnumerationHandler method for getting the name of the state which a variable assignment represents.

When `GameBuilder` creates a `SymbolicEnergyGame`, it will create all the rules of the game, in a way similar to that seen in Section 5.2.4. This is shown in Listing 5.11. Once a `SymbolicEnergyGame` has been built, we are ready to compute a greatest fixed point for the game. This fixed point algorithm, shown in Listing 5.12 corresponds to the one presented in Algorithm 1 of Section 2.3. As previously mentioned, BuDDy works with a custom garbage collector. Therefore any BDDs which are no longer used in the end of the loop have their references deleted, in order to free any unused nodes in BuDDy. This can be seen by explicit calls in the code to `DelRef`.

```
1  /// <summary>
2  /// Builds an Energy Game using BDDs. The Energy Game is used for symbolic ↩
       computation
3  /// </summary>
4  /// <returns>A Symbolic Energy Game.</returns>
5  public SymbolicEnergyGame BuildBDDEG()
6  {
7    //Create an Enumeration Handler and initialize it
8    EnumerationHandler eh = new EnumerationHandler(Statesets, Weightsets, wMax, wMin, ↩
         greatestAdd, greatestSub);
9
10   BDD Erules = BUDDY.BDDFalse();
11   BDD Urules = BUDDY.BDDFalse();
12   int i = 0;
13   foreach (Rule rule in Rules)
14   {
15     if (rule.Universal)
16     {
17       Urules = Urules | MakeRuleBDD(rule, eh);
18     }
19     else
20     {
21       Erules = Erules | MakeRuleBDD(rule, eh);
22     }
23   }
24   return new SymbolicEnergyGame(eh, Erules, Urules);
25 }
```

Listing 5.11: The method used by `GameBuilder` to build a `SymbolicEnergyGame`.

```
BDD w;
BDD wn = BUDDY.BDDTrue() & invalidValues; //All invalid values listed, and negated, meaning invalid values (outside bounds) lead to 0
BDD w_;
BDD exist;
BDD forall;
Steps = 0;
Console.WriteLine("Computing Winning BDD..");
do
{
  //Update W
  w = wn;
  //Create Wxprime, by replacing all variables with prime variables
  w_ = w;
  foreach (BDDPair pair in pairs)
  {
    w_ = BUDDY.Replace(w_, pair);
  }
  //Existential quantification, all prime variables
  exist = BUDDY.Exist(Erules,primesVar).Implies(BUDDY.AppExist(Erules, w_, BUDDY.Operator.AND, primesVar));
  //Universal Quantification, all primes
  forall = BUDDY.AppAll(Urules, w_, BUDDY.Operator.IMP, primesVar);
  //Update Wn
  wn = w & exist & forall;
  w.DelRef();
  exist.DelRef();
  forall.DelRef();
  if (Steps % 10 == 0 && Steps != 0)
    Console.WriteLine("Iteration: " + Steps);
  Steps++;
} while (w != wn);
BUDDY.PrintLabelled("Win", wn); //Dump the winning BDD as .dot
WinningBDD = wn;
w.DelRef();
w_.DelRef();
primesVar.DelRef();
if(BUDDY.NodeCount(wn) == 0)
Console.WriteLine("Winning BDD found in " + Steps + " iterations.");
else
Console.WriteLine("After " + Steps + " iterations, no winning space was found.");
return wn;
```

Listing 5.12: The main part of the fixed point algorithm in `SymbolicEnergyGame`.

CHAPTER 6

EGGS - A MULTIWEIGHTED ENERGY GAMES TOOL

In this chapter we will demonstrate the tool and what is possible with it.

6.1 Features of EgGS

When EgGS is first launched, the user is presented with two windows; a console and a main window similar to that seen in Figure 6.1. The main window is primarily made up of a text editor, which is used for writing LEG syntax. After expresseing a game, initially, only the `Compute!` button is enabled. Once the `Compute!` button has been pressed, a greatest fixed point will be computed of the game described by the syntax in the text editor. For details on the LEG syntax and how to express games in LEG, please refer to Chapter 3.

The text editor, as seen in Figure 6.1, highlights keywords, comments and values. Once the text editor holds a valid syntax, the greatest fixed point can be computed by clicking the `Compute!` button, as seen in Figure 6.1. The console shows information about the current computational step.

Once a winning BDD has been computed, the previously disabled buttons `Erules BDD`, `Urules BDD`, `Show Winning List`, `Show Strategy` and `Show BDD`, are enabled.

Starting from the far right, the `Show BDD` will show a BDD of the greatest fixed point as seen in Figure 6.2. The values represented for weights in this BDD are subject to the offset value, as described in Section 4.2.2. Note that the rendering of the shown BDDs is currently dependent on the dot render of Graphviz[1], which comes packaged with EgGS.

The button `Show Strategy`, displays the window shown in Figure 6.3. This strategy simply lists what actions (rules) stay in the winning space given a weighted configuration. In case the rules have not been named, the strategy is instead expressed as a transition from a weighted configuration to another weighted configuration - implicitly expressing the rule.

To see the list of all weighted configurations in the winning space, the menu seen in Figure 6.4 can be shown by clicking the `Show Winning List`. Note that this may require a high amount

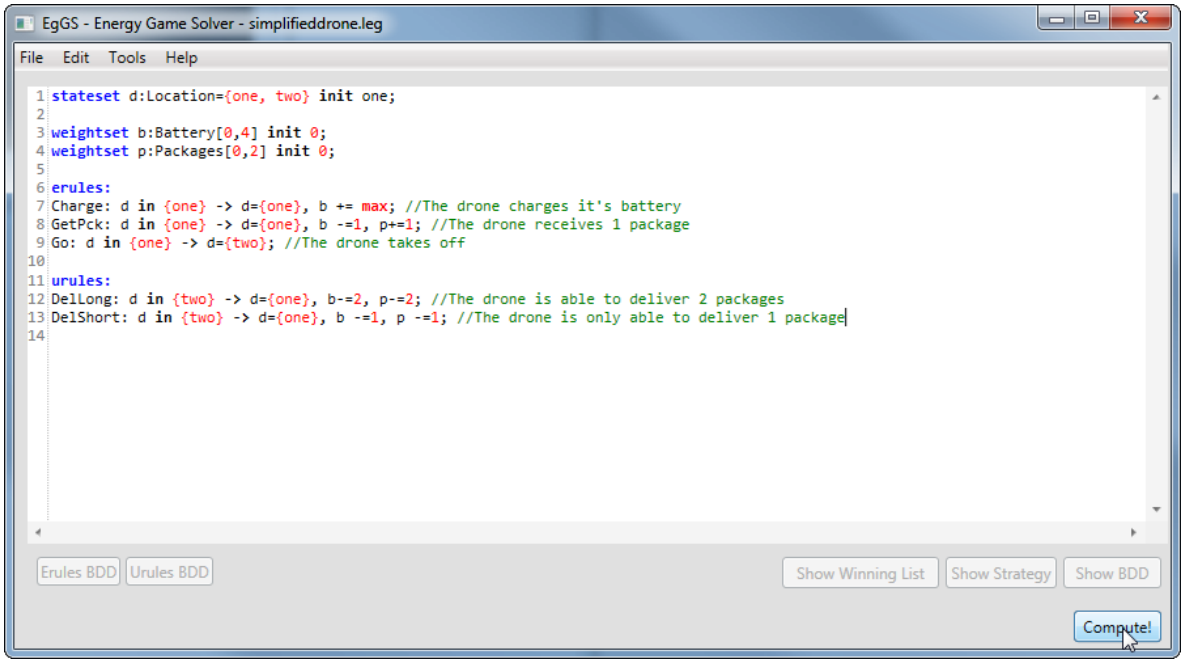


Figure 6.1: The compute button is located in the bottom right corner of the main window.

of rendering time for very large games.

The two buttons `Erules BDD` and `Urules BDD`, each display a BDD similar to the one seen in Figure 6.2, but exclusively for existential rules or universal rules, respectively. These BDDs might be helpful when looking for errors in encoded rules.

It is also possible to compute a fixed point without the symbolic representation, using the explicit algorithm. The explicit algorithm can be selected as seen in Figure 6.5. The console will still register information about the process, but the weighted configurations of the explicit computation will also be displayed in the console.

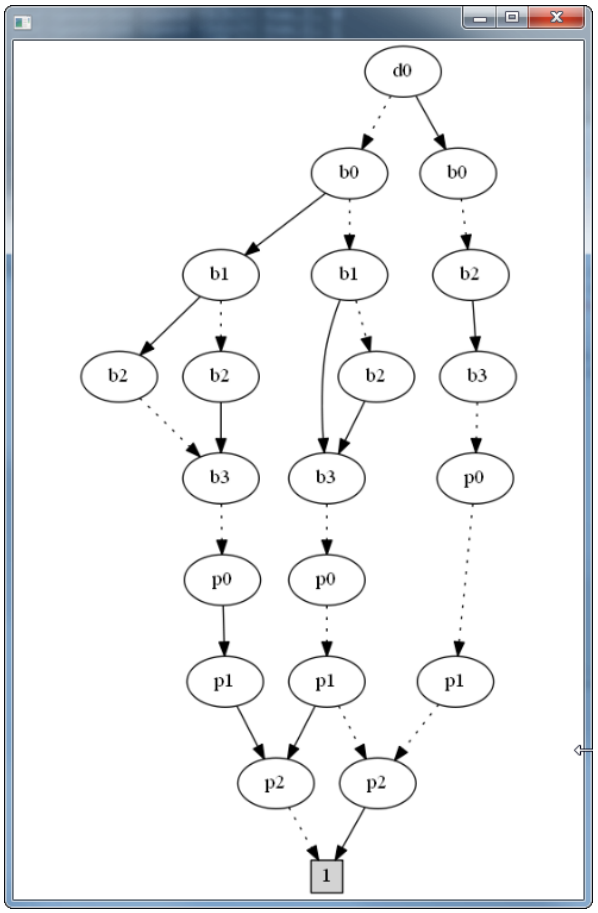


Figure 6.2: Each satisfying assignment to the winning BDD, represents a winning weighted configuration.

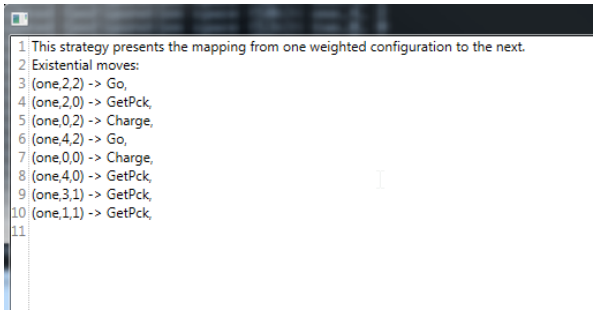


Figure 6.3: A strategy, as displayed through EgGS.

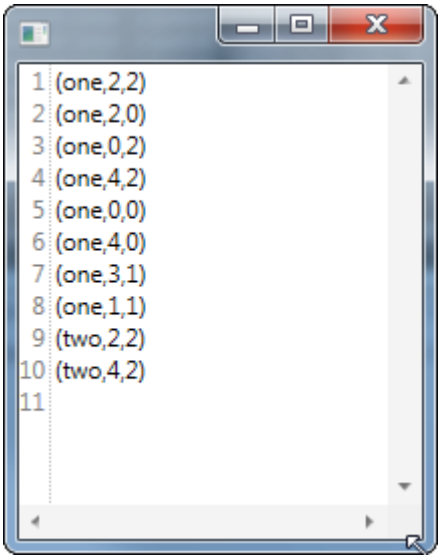


Figure 6.4: A list of all weighted configurations that are part of the winning space.

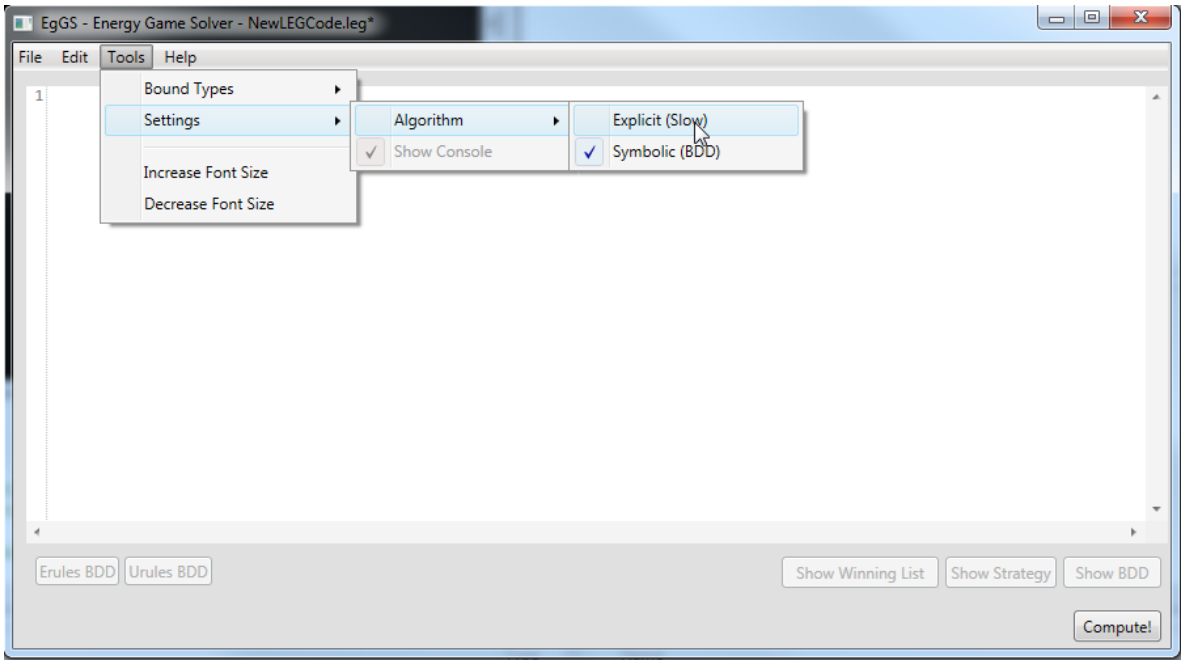


Figure 6.5: It is possible to select an explicit algorithm in EgGS.

6.2 Extensions to EgGS

The EgGS tool is not without shortcomings, and some features of LEG are currently unsupported. A full suite of analytic tools could be added to accommodate the analytic work associated with synthesizing energy strategies. Most the desirable features are mainly GUI related *e.g.* better representation and presentation of results, such as exporting results in different formats. However, as we have shown, the core functionality of the tool is robust, and ready for immediate use.

- Currently the symbolic encoding does not support rules with weight assignments involving a weight variable. An example of such a rule could be

  $$\texttt{x in } \{\texttt{one}\},\ \texttt{y in } \{\texttt{alpha}\} -> \texttt{x} = \{\texttt{two}\}, \texttt{b} += 2 * \texttt{a};$$

  , where `a` and `b` are both weight variables. As weight assignments involving values of other weights is a strength of the LEG syntax, it should be supported in the symbolic encoding.

- The BuDDy framework supports the export and import of BDDs. Thus, it is desirable to support the export of the various BDDs generated by EgGS. Saving the computed winning BDD, will also allow the strategy to be synthesized without recomputing that BDD.

- By simulating an energy game as a step-by-step game between the user as the controller, and the environment controlled by EgGS, users may come to better understand their energy games and how a strategy may fail.

- As mentioned in Section 2.4, there are several variants of energy games. While the support for strong and weak bounds would be most efficiently implemented through the LEG syntax, EgGS should also implement support for strong and weak bounds.

- There are several ways to present a strategy. As mentioned in Section 2.2, the MTIDD structure might be an interesting way to present a strategy. The current presentation of strategies is needlessly big, and certainly better presentations exist. A strategy may also be represented directly as a MTBDD, which is supported by BuDDy.

- Currently the representation of timed games require explicit syntax to express them in LEG. Until LEG provides support for guards on rules, and invariants on configurations, EgGS may implement a tool to transform a LEG syntax and some separately specified constraints, into a LEG syntax representing the discreetly timed game.

- BuDDy is initialized with two values; a value for the number of preallocated nodes and a node caching value. It would be beneficial to either allow the user to specify these values or improve the dynamic adjustment of these values for any specified energy game.

- A consideration for computation is the option of allowing the reordering of variables. Variable reordering can be an expensive operation, but for very large games with many iterations, it might provide a benefit. Additional experimentation is required to establish whether this would provide a benefit or not.

- Generating a graph displaying the energy game described by the syntax would be beneficial.

CHAPTER 7

EXPERIMENTS

In this section we will evaluate the efficiency of the algorithms used in this version of the tool. We will compare the efficiency to the explicit representation of games, and comment on the perceived complexities of the symbolic approach.

7.1 Setup

This section details how the tests were conducted. The test equipment is first presented, followed by details about the test cases.

All the tests in were carried out on a machine with an Intel Core i7-3612QM CPU, running at 2.10GHz, with 6 GB of DDR3 memory running at a bandwidth 800MHz. The i7-3612QM has an L3 cache of 6MB, 4 L2 caches of 256KB, and 8 L1 caches of 32KB.

Memory measurements were performed using a profiling tool, with a memory snapshot of each stage in computation. Time measurements are done with the C♯ class `StopWatch`, which measures elapsed wall-clock time, with an accuracy to fractions of milliseconds. This time was also saved during each stage of the computation.

Memory results are dependent on a few uncontrollable variables. As the BDD operations are memory-intensive operations, it should be noted that performance is very dependent on the number of calls to allocate memory and the number of calls to BuDDy's own garbage collector. By design in our implementation, a game will preallocate a larger amount of BuDDy nodes for larger games. This value is currently set to allocate a number of nodes corresponding rougly to 150MB. This number is manageable by a majority of modern PCs, and results in a large improvement to the performance of BuDDy. For smaller games, the tool will only allocate a few MBs of memory initially.

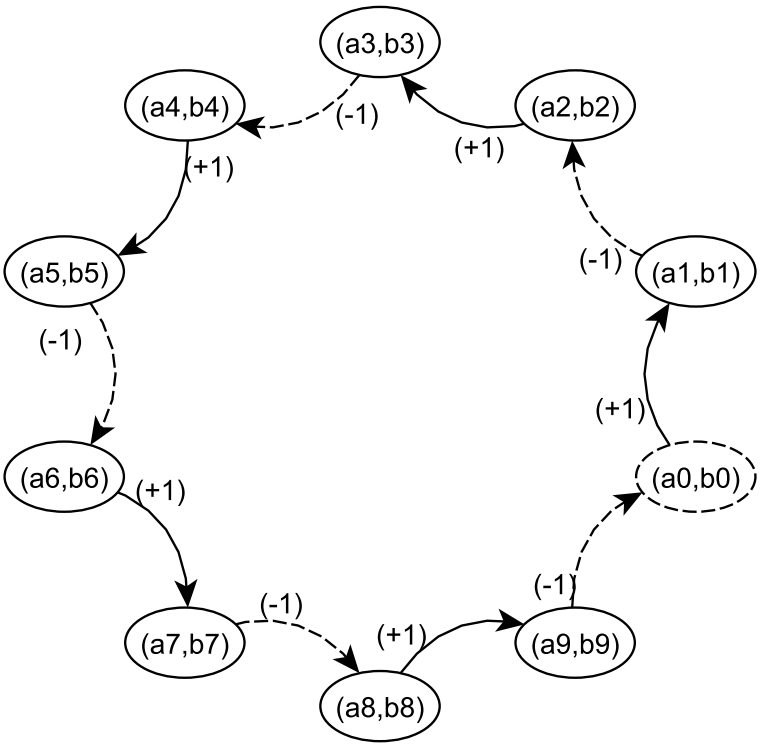


Figure 7.1: Game for states and statesets tests.

7.1.1 Test cases

The tests are based on two base cases, as seen in Listing 7.1 and in Listing 7.2. These base cases are scaled up to provide a wide array of different test cases.

The LEG syntax in Listing 7.1 have been scaled up by increasing the number of statesets and the number of states in each stateset. Each such scaling also increases the number of rules to provide the same circular energygame as seen in Figure 7.1. These test cases are named MsNss, where M and N represent a number.

The LEG syntax in Listing 7.2 results game seen in Figure 7.2.
This game is scaled up in two ways. The first fashion in which the game is scaled, is by an increasing upper bound of both weightsets. Each case of this type will have the upper bound increased by a multiplier. This effectively increases the number of weighted configuration represented. These test cases are named `owNx`, where `Nx` is the multiplier. In addition to increasing the upper bound, the other variant also scales the values added and subtracted in the rules. These test cases are named `warNx`, where `Nx` is the multiplier.

For all cases, four values are measured: The memory used during construction, the memory used during the computation of a fixed point, the time spent on construction and the time spent on computation. In each case, these values are measured for both the explicit and symbolic representation. The time spent on construction was added to the time spent computing to

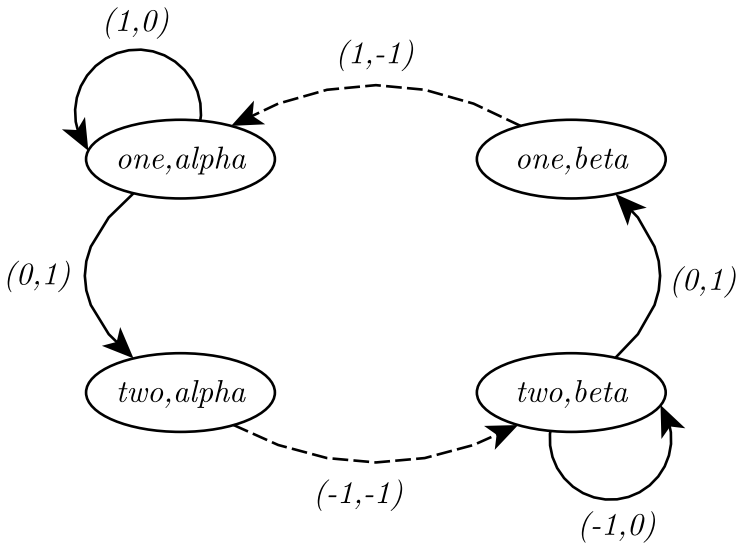


Figure 7.2: Game for weightsets tests.

```
1  stateset a:A={a0,a1,a2,a3,a4,a5,a6,a7,a8,a9} init a0;
2  stateset b:B={b0,b1,b2,b3,b4,b5,b6,b7,b8,b9} init b0;
3  weightset x:X[0,1] init 0;
4  <a in {a0},b in {b0} -> a={a1},b={b1},x+=1>;
5  [a in {a1},b in {b1} -> a={a2},b={b2},x-=1];
6  <a in {a2},b in {b2} -> a={a3},b={b3},x+=1>;
7  [a in {a3},b in {b3} -> a={a4},b={b4},x-=1];
8  <a in {a4},b in {b4} -> a={a5},b={b5},x+=1>;
9  [a in {a5},b in {b5} -> a={a6},b={b6},x-=1];
10 <a in {a6},b in {b6} -> a={a7},b={b7},x+=1>;
11 [a in {a7},b in {b7} -> a={a8},b={b8},x-=1];
12 <a in {a8},b in {b8} -> a={a9},b={b9},x+=1>;
13 [a in {a9},b in {b9} -> a={a0},b={b0},x-=1];
```

Listing 7.1: Case 1.

give the total time. The maximum memory of either construction or computation was used in the results. The results displayed in the following sections are all shown with total time and the maximum memory used.

7.1.2 Limitations

Comparing the symbolic and the explicit approach directly is difficult - the scalability of the symbolic representation is best showed for large examples. However, the explicit implementation is extremely slow for such large games. The most time-consuming symbolic test, is computed in roughly 55 minutes. As this is the maximum, every test was set to timeout after one hour. In the case of a timeout, no discernible conclusions can be made from the data of the explicit algorithm. This timeout is necessary, because as showed in [8], it takes several

```
stateset x:X={one, two} init one;
stateset y:Y={alpha, beta} init alpha;

weightset a:A[0,4] init 0;
weightset b:B[0,1] init 0;

erules:
x in {one}, y in {alpha} -> a+=1;
x in {two}, y in {beta} -> a-=1;
x in {one}, y in {alpha} -> x={two}, b+=1;
x in {two}, y in {beta} -> x={one}, b+=1;

urules:
x in {two}, y in {alpha} -> y={beta}, a-=1, b-=1;
x in {one}, y in {beta} -> y={alpha}, a+=1, b-=1;
```

Listing 7.2: Case 2.

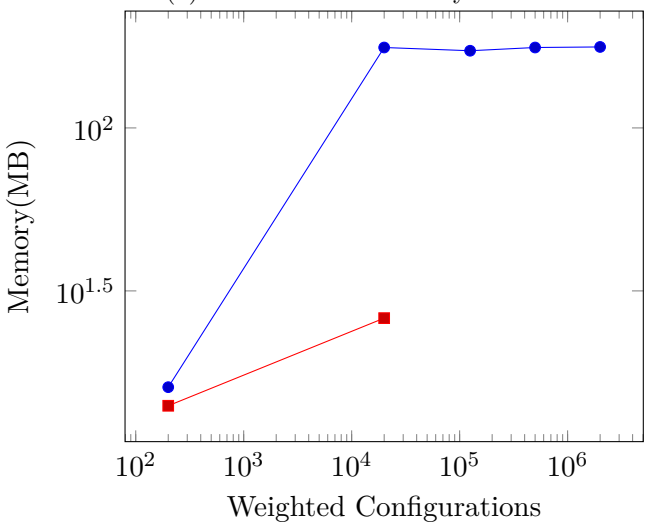
hours to compute a winning space for games with a weighted configuration of about 10^5, and even days when approaching a magnitude of 10^5 and 10^6.

7.2 Data

Shown in this section is the data from the test results. Time is shown in minutes and seconds, but the raw data can be seen in Appendix D.

Test case	Weighted Configurations	Memory by symbolic (MB)	Memory by explicit (MB)
10s2ss	200	16.02	14.05
100s2ss	20000	176.5	26.09
250s2ss	125000	172.5	*Timeout*
500s2ss	500000	176.5	*Timeout*
1000s2ss	2000000	177.2	*Timeout*

(a) Case **Ms2ss**: Memory.

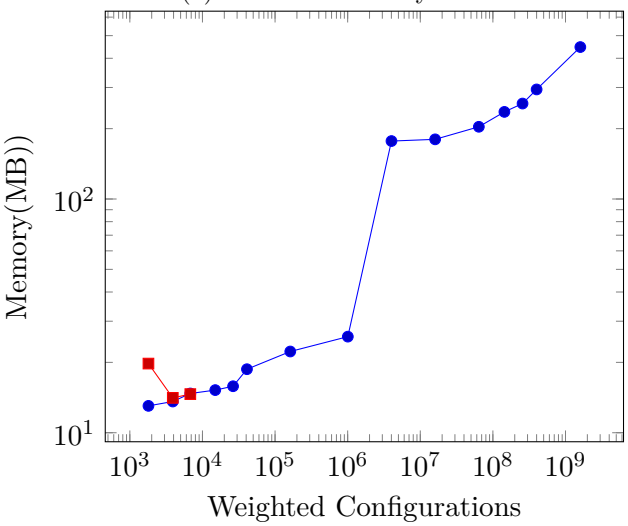


(b) Plot of memory for `Ms2ss`, found in Table 7.1a.

Table 7.1: Test case `Ms2ss`, memory results.

Test case	Weighted Configurations	Memory by symbolic (MB)	Memory by explicit (MB)
ow10x	1804	13.03	19.77
ow15x	3904	13.6	14.1
ow20x	6804	14.72	14.63
ow30x	15004	15.22	*Timeout*
ow40x	26404	15.81	*Timeout*
ow50x	41004	18.7	*Timeout*
ow100x	162004	22.26	*Timeout*
ow250x	1005004	25.77	*Timeout*
ow500x	4010004	177	*Timeout*
ow1000x	16020004	180	*Timeout*
ow2000x	64040004	203.7	*Timeout*
ow3000x	144060004	235.9	*Timeout*
ow4000x	256080004	256.1	*Timeout*
ow5000x	400100004	294.3	*Timeout*
ow10000x	1600200004	446.8	*Timeout*

(a) Case `ow`: Memory.

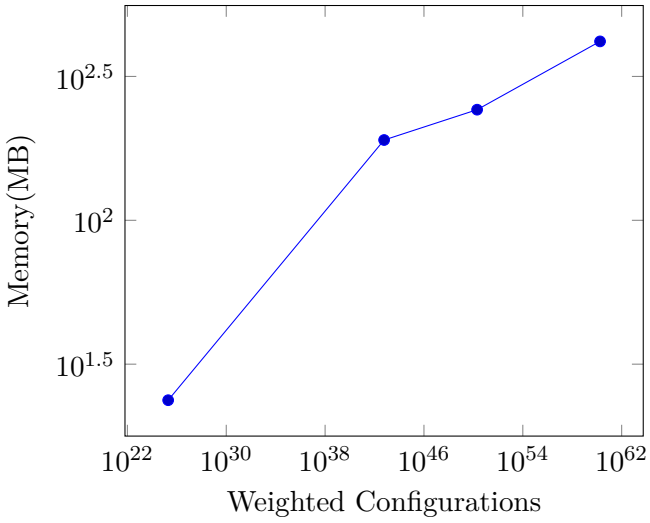


(b) Plot of memory for `ow`, found in Table 7.2a.

Table 7.2: Test case `ow`, memory results.

Test case	Weighted Configurations	Memory by symbolic (MB)	Memory by explicit (MB)
10s25ss	2×10^{25}	23.7	*OOM*
50s25ss	5.96046×10^{42}	190.1	*OOM*
100s25ss	2×10^{50}	242.3	*OOM*
250s25ss	1.77636×10^{60}	418.5	*OOM*
500s25ss	5.96046×10^{67}	*OOM*	*OOM*
1000s25ss	2×10^{75}	*OOM*	*OOM*

(a) Case `Ms25ss`: Memory.

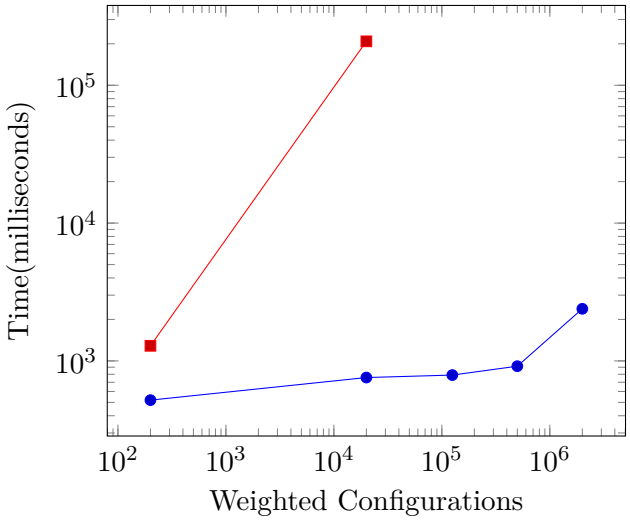


(b) Plot of memory for `Ms25ss`, found in Table 7.3a.

Table 7.3: Test case `Ms25ss`, memory results.

Test case	Weighted Configurations	Time by symbolic	Time by explicit
10s2ss	200	0.5192873 s	1.287193 s
100s2ss	20000	0.7574245 s	3 min 28.4298882 s
250s2ss	125000	0.7894276 s	$Timeout$
500s2ss	500000	0.9154329 s	$Timeout$
1000s2ss	2000000	2.3884742 s	$Timeout$

(a) Case `Ms2ss`: Time.

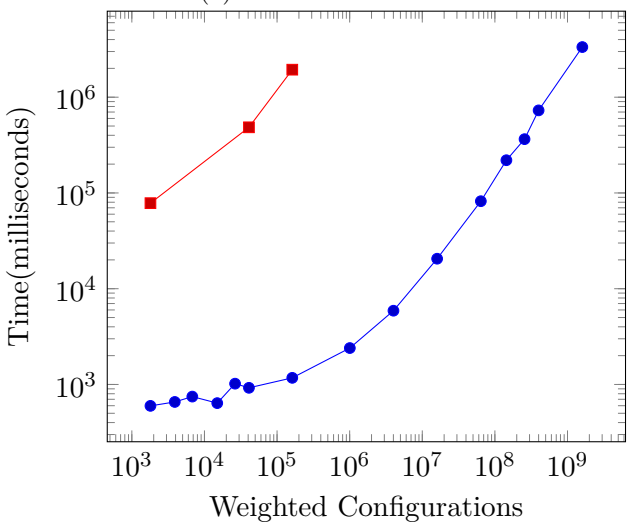


(b) Plot of time for `Ms2ss`, found in Table 7.4a.

Table 7.4: Test case `Ms2ss`, time results.

Test case	Weighted Configurations	Time by symbolic	Time by explicit
ow10x	1804	0.597709 *s*	1 *min* 18.2604205 *s*
ow15x	3904	0.6567422 *s*	8 *min* 4.8949852 *s*
ow20x	6804	0.7469507 *s*	32 *min* 18.694073 *s*
ow30x	15004	0.638447 *s*	*Timeout*
ow40x	26404	1.0193067 *s*	*Timeout*
ow50x	41004	0.9224052 *s*	*Timeout*
ow100x	162004	1.1724107 *s*	*Timeout*
ow250x	1005004	2.4002739 *s*	*Timeout*
ow500x	4010004	5.8915896 *s*	*Timeout*
ow1000x	16020004	20.5681978 *s*	*Timeout*
ow2000x	64040004	1 *min* 22.0163743 *s*	*Timeout*
ow3000x	144060004	3 *min* 39.7417351 *s*	*Timeout*
ow4000x	256080004	6 *min* 4.6073954 *s*	*Timeout*
ow5000x	400100004	12 *min* 8.5975853 *s*	*Timeout*
ow10000x	1600200004	55 *min* 39.70168 *s*	*Timeout*

(a) Case `ow`: Time.

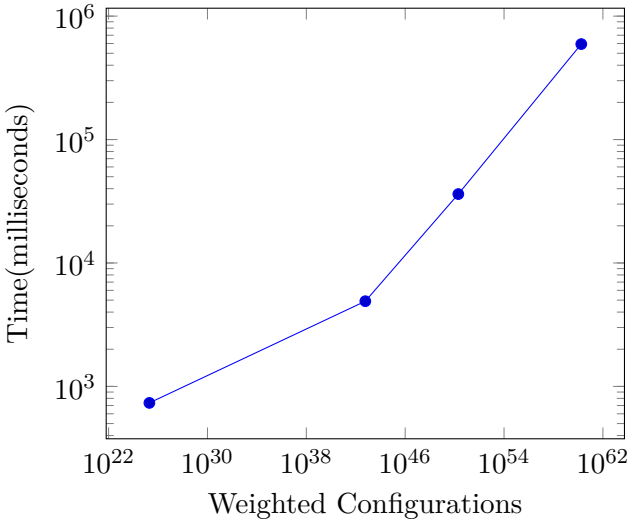


(b) Plot of time for `ow`, found in Table 7.5a.

Table 7.5: Test case `ow`, time results.

Test case	Weighted Configurations	Time by symbolic	Time by explicit
10s25ss	2×10^{25}	0.7351674 s	OOM
50s25ss	5.96046×10^{42}	4.9005141 s	OOM
100s25ss	2×10^{50}	36.1657469 s	OOM
250s25ss	1.77636×10^{60}	9 min 53.5291146 s	OOM
500s25ss	5.96046×10^{67}	OOM	OOM
1000s25ss	2×10^{75}	OOM	OOM

(a) Case `Ms25ss`: Time.



(b) Plot of time for `Ms25ss`, found in Table 7.6a.

Table 7.6: Test case `Ms25ss`, time results.

7.3 Discussion

When looking at all the data it is immediately clear that the symbolic representation is not only faster, but able to represent much larger games. This is of course due to the symbolic representation - where the explicit representation lists and checks each weighted configuration in the game, the symbolic representation merely adds a boolean variable whenever needed. This means that the exponential blow-up seen in the explicit representation is not present in the symbolic.
In the symbolic representation, the time required to find a winning space is more dependent on the rules themselves, rather than the number of possible weighted configurations. This also means that the algorithm terminates much earlier if no strategy is possible.
To support this claim, consider the case `ow100x`. For this case a winning space does indeed exist, and it is found in only 1.17 seconds. However, if we were to design the rule in such a way that it leads to a less complex BDD, we would require a lot fewer iterations.

```
1  stateset x:X={one, two} init one;
2  stateset y:Y={alpha, beta} init alpha;
3
4  weightset a:A[0,400] init 0;
5  weightset b:B[0,100] init 0;
6
7  erules:
8  x in {one}, y in {alpha} -> a+=401;
9  x in {two}, y in {beta} -> a-=401;
10 x in {one}, y in {alpha} -> x={two}, b+=401;
11 x in {two}, y in {beta} -> x={one}, b+=401;
12
13 urules:
14 x in {two}, y in {alpha} -> y={beta}, a-=401, b-=401;
15 x in {one}, y in {beta} -> y={alpha}, a+=401, b-=401;
```

Listing 7.3: `ow100x` modified to have no winning strategy.

Consider the alterations made to `ow100x` in Listing 7.3. In this case, it should be clear that no winning strategy is possible - every rule in the game breaks the bound no matter what. As such the rules do not create any new nodes in a BDD - in the quantification step no assignments satisfies the BDDs describing the rules. And as such, the algorithm terminates immediately after two iterations, in only 0.44 seconds.

It should be noted that due to the memory snapshots that were taking along the process, some of the lower time results may be slightly skewed. This is due to the memory snapshot, taking an average of 550ms. This value has little impact on the comparison of the algorithms however, as this delay is present for all datasets, both in the explicit and symbolic representation. Each dataset does have slight fluctuations, since only one test was done on each test-case, but this does not seem to impact the interpretation of the data.

7.3.1 Space

The results on memory testing are displayed in Tables 7.1a, 7.2a and 7.3a. Some of these results are marked as *OOM*, denoting that the system encountered an Out Of Memory

exception. Likewise, some results are marked as *Timeout*, if the system exceeded the 1 hour time limit.

A common pattern for all the memory test case is a sudden jump in memory consumption - at places it would seem that the symbolic representation consumes more memory than the explicit approach. This is mostly due to the desire for speed in the symbolic representation. As mentioned earlier, BuDDy works with a custom garbage collector, and allocating often and garbage collecting often can have a severe impact on performance. Depending on the estimated size of a game, and the complexity of the rules, a certain cache size of BuDDy and initial allocated memory is set. It is an ongoing effort to improve the way the tool predicts the amount of memory needed. However, the largest pre-allocated memory in BuDDy is roughly 150MB. This is apparent from the memory data points being more consistent when the memory exceeds this value.

The `Ms2ss` test cases, seen in Table 7.1, maintained 2 statesets and 1 weightset while scaling the number of states in each stateset up. The sudden jump in the symbolic algorithms, is assumed to be caused by BuDDy pre-allocating a larger amount of nodes for large games.

The `ow` test cases, seen in Table 7.2, maintained the same game graph, but adjusted the upper bound of the weightsets. The first thing worth noting is that these test cases, even though they express more weighted configuration than the `Ms2ss` test cases, use less memory for a comparable number of weighted configurations. For both algorithms, it seems encoding large weightsets require less memory than encoding large statesets. While it seems that the symbolic representation is more efficient, the results from the explicit algorithm unfortunately timed out. Again we observe the sudden spike in memory usage between test case $ow250x$ and test case $ow500x$.

Overall the symbolic approach is far less dependant on the constant of the weightsets. The memory representation is directly dependent on the number of nodes in the BDDs constructed. The number of nodes in a BDD in buddy, is influenced partly by the number of variables needed to represent a game, but also the amount, and complexity of the rules involving those variables.

7.3.2 Time

The results on time testing are displayed in Tables 7.4a, 7.5a and 7.6a. Overall the symbolic approach is far superior to the explicit approach. In Table 7.5a we see how the symbolic representation finds a winning space in fractions of seconds, compared to the explicit approach, and how it handles large examples with ease.
In [8], we saw how even moderately sized games took hours to compute, as seen in Figure 7.3 It should be noted that the results shown there cannot be directly compared, as they were conducted on another test case. It is however a good indication of the exponential blow-up seen in the explicit approach.

The `Ms2ss` test cases, seen in Table 7.4, show a clear difference between the symbolic algorithm and the explicit algorithm. Even in the simplest of the test cases, the $10s2ss$ case with a mere

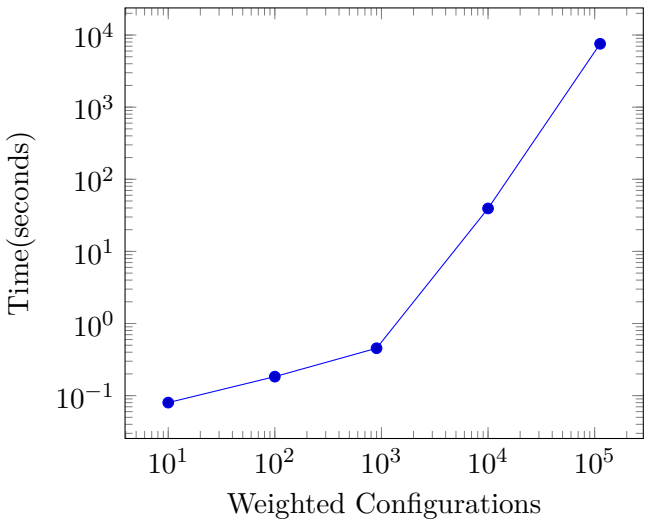


Figure 7.3: The slow times of the explicit approach, as shown in [8].

200 weighted configurations, it is clear that the symbolic algorithm uses substantially less time to compute a fixed point, compared to the explicit algorithm. Even at a higher amount of weighted configurations, where the explicit exceeds the 1-hour time out, the symbolic algorithm takes mere seconds to find the winning space.

The `ow` test cases, seen in Table 7.2, also demonstrate a clear advantage in the symbolic algorithm, compared to the explicit algorithm. Already at the $ow30$ test case, with about 15.000 weighted configurations, the explicit algorithm requires over an hour of computation time. The symbolic algorithm does not reach any computation time similar until the $ow10000x$ case, which has 1.6 billion weighted configurations.

7.3.3 Complexity

As mentioned briefly in Section 4.1, the complexities of operations on BDDs, scale linearly with the size of the BDDs. As such the complexity of an energy game, is directly influenced by the efficiency of the operations on BDDs. During the fixed point algorithm, we are only using calls to Apply and Restrict, which both scale linearly with the number of nodes in the BDD. Quantification is, as mentioned, implemented using a call to Apply, and two calls to Restrict.
In the very worst-case, the blow-up of a BDD may be exponential as well. However, in this case, the average case seems to scale polynomially.
The worst case of BDD complexity, would be a game, where the game is fully connected between all configurations, and where the weights assignments in these rules, all result in weighted configurations being in the winning space, and branching the BDD out for every single possible weighted configuration. Such a BDD is highly unlikely.

CHAPTER 8

CONCLUSION

We have shown an approach to encode energy games, written with LEG syntax, symbolically. A symbolic fixed point algorithm have been presented. We have implemented the transformation to symbolic encoding and the symbolic algorithm in the tool EgGS. The tool, EgGS has been used to test both the symbolic algorithm and explicit algorithm. The tests clearly indicate that the symbolic algorithm is superior in computation time. The tests also indicate that the symbolic algorithm consumes less memory compared to the explicit algorithm when dealing with large amounts of weighted configurations.
The process of synthesizing a strategy using the symbolic algorithm is mainly dependent on the operations on binary decision diagrams. As such a strategy can be found in polynomial time in the average case. The explicit approach has an exponential growth relative to the number of weighted configurations, which prevents the representation of larger games.
It has been shown in Section 4.2 how the syntax of LEG, can be encoded to describe a game through boolean variables and expressions. The binary decision diagram data structure has been applied to compactly encode the boolean expressions. With the usage of the BDD data structure, several important operations during the symbolic fixed point computation, have been made easy.
We have presented the features and the implementation of EgGS in detail, and elaborated on how it can be used to generate strategies. In regards to the usability and overall utility of the tool, more features could be desired, however, we have shown that the core implementation is both robust and usable.
It has been demonstrated that LEG syntax is capable of representing a discretely timed game and a strategy for a timed example has been presented, without any changes to complexity.
What follows from this project, is a software tool which can accurately and efficiently generate an energy strategy even for complex energy games expressed in LEG.

LIST OF FIGURES

LIST OF TABLES

BIBLIOGRAPHY

[1] Graphviz. `http://www.graphviz.org/`. [Online; accessed June-2014].

[2] Windows presentation foundation. `http://msdn.microsoft.com/en-us/library/ms754130(v=vs.110).aspx`. [Online; accessed May-2014].

[3] L. Aceto, A. Ingolfsdottir, K. G. Larsen, and J. Srba. *Reactive Systems: Modelling, Specification and Verification*. Cambridge University Press, 2007.

[4] H. R. Andersen. An introduction to binary decision diagrams. Lecture Notes, 1999.

[5] P. Bouyer, U. Fahrenberg, K. G. Larsen, N. Markey, and J. Srba. Infinite runs in weighted timed automata with energy constraints. In F. Cassez and C. Jard, editors, *FORMATS*, volume 5215 of *Lecture Notes in Computer Science*, pages 33–47. Springer, 2008.

[6] T. Brazdil, P. Jancar, and A. Kucera. Reachability games on extended vector addition systems with states.

[7] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, Aug. 1986.

[8] M. V. Carlsen and R. S. Jacobsen. Multiweighted energy games and the leg language. Technical report, Aalborg University, 2014.

[9] J. Chaloupka. Z-reachability problem for games on 2-dimensional vector addition systems with states is in p. In *RP'10 Proceedings of the 4th international conference on Reachability problems*, pages 104–119.

[10] K. Chatterjee and L. Doyen. Energy parity games.

[11] K. Chatterjee, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Generalized mean-payoff and energy games. In *Proceedings of FSTTCS'10*, pages 505–516. LIPIcs, 2010.

[12] A. Degorre, L. Doyen, R. Gentilini, J.-F. Raskin, and S. Torunczyk. Energy and mean-payoff games with imperfect information. In *CSL'10/EACSL'10 Proceedings of the 24th international conference/19th annual conference on Computer science logic*, pages 260–274.

[13] U. Fahrenberg, K. G. Larsen, J. Srba, and L. Juhl. Energy games in multiweighted automata. In *ICTAC'11 Proceedings of the 8th international conference on Theoretical aspects of computing*.

[14] J. Lind-Nielsen. Buddy. `http://buddy.sourceforge.net/manual`. [Online; accessed May-2014].

[15] MonoProject. Mono project. `http://www.mono-project.com/`. [Online; accessed May-2014].

[16] T. Parr. Antlr. `http://www.antlr.org`. [Online; accessed January-2014].

[17] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific J. Math*, 5(2):285–309, 1955.

APPENDIX A

SYNTAX AND SEMANTICS OF LEG

In this appendix we present the syntax and semantics of LEG. This is an excerpt of the previous report [8].

A.1 Abstract Syntax

The abstract syntax M of LEG is given by:

$$M ::= \underline{\textbf{stateset}}\ SSet\ \underline{\textbf{weightset}}\ WSet\ \underline{\textbf{in rules}}\ ERules; URules,$$

and we define the set **MODELS** to be the set of all models described by this syntax.

The collection of all statesets is given by

$$SSet ::= s_1 : S_1 \texttt{init} a_1; \ldots; s_n : S_n \texttt{init} a_n$$

$M ::= \underline{\textbf{stateset}}\ SSet\ \underline{\textbf{weightset}}\ WSet\ \underline{\textbf{in rules}}\ ERules; URules$
$SSet ::= s_1 : S_1 \texttt{init} a_1; \ldots; s_n : S_n \texttt{init} a_n$, where S_i is a finite set and $a_i \in S_i$
$WSet ::= w_1 : W_1 \texttt{init} v_1; \ldots; w_k : W_k \texttt{init} v_k$, where $W_i ::= \{j \in \mathbb{N} \mid l_i \leq j \leq u_i\}$ and $v_i \in W_i$
$ERules ::= <Rule_1>, \ldots, <Rule_n>$
$URules ::= [Rule_1], \ldots, [Rule_n]$
$Rule ::= Pre- > Upd$
$Pre ::= s_1 \in A_1, \ldots, s_n \in A_n$, where $A_i \subseteq S_i$
$Upd ::= s_1 :\in SExp_1; \ldots; s_n :\in SExp_n; w_1 := WExp_1; \ldots; w_k := WExp_k$
$SExp_i ::= s_i \mid A_i$, where $A_i \subseteq S_i$
$WExp_j ::= w_l \mid WExp\ op\ WExp \mid n$ where $op \in \{+, -, *\}$ and $n \in \mathbb{N}$

Table A.1: Abstract Syntax of LEG.

where $S_1, \ldots, and\ S_n$ are finite sets of states, s_i is the statevariable and the initial state is $a_i \in S_i$. We define the set **SSETS** to be the set of all statesets described by this syntax.

$$WSet ::= w_1 : W_1 \texttt{init} v_1; \ldots; w_k : W_k \texttt{init} v_k$$

is a collection of weightsets, where W_i is a set of integers from some lower bound l_i to some upper bound u_i, i.e. $W_i ::= \{j \in \mathbb{N} \mid l_i \leq j \leq u_i\}$. The initial value of the weightvariable w_i is $v_i \in W_i$. We define the set **WSETS** to be the set of all weightsets described by this syntax.

The initial weighted configuration c_0 is given by $(a_1, \ldots, a_n, v_1, \ldots v_k)$.

$$ERules ::= <Rule_1>, \ldots, <Rule_n>$$

is a list of existential rules. We define the set **ERULES** to be the set of all existential rules described by this syntax.

$$URules ::= [Rule_1], \ldots, [Rule_n]$$

is a list of universal rules. We define the set **URULES** to be the set of all universal rules described by this syntax.

Every rule $Rule ::= Pre- > Upd$ consists of preconditions Pre and an update Upd. We define the set **RULES** to be the set of all rules described by this syntax.

A precondition is a specification that a state variable s_i must be in some set of states A_i, where A_i must be a subset of the stateset S_i for a rule to be enabled. A precondition is given by

$$Pre ::= s_1 \in A_1, \ldots, s_n \in A_n,$$

where $A_i \subseteq S_i$. We define the set **PRE** to be the set of all preconditions described by this syntax.

An update is the assignment of the state variable s_i to be in some state expression $SExp_i$ and for some weight w_j to be assigned the value specified by the weight expression $WExp_j$, i.e.

$$Upd ::= s_1 :\in SExp_1; \ldots; s_n :\in SExp_n; w_1 := WExp_1; \ldots; w_k := WExp_k.$$

We define the set **UPD** to be the set of all updates described by this syntax.

A state expression $SExp_i$ is either the state s_i (i.e. the state would remain the same) or a set of states A_i, which is a subset of S_i, i.e.

$$SExp_i ::= s_i \mid A_i,$$

where $A_i \subseteq S_i$. We define the set $\mathbf{SEXP}_i$ to be the set of all stateexpressions described by this syntax.

A weight expression is either some weightvariable w_l or some natural number n or the arithmetic expression of two weight expressions, limited to the arithmetic operators for multiplication, subtraction and addition, i.e.

$$WExp_j ::= w_l \mid WExp\ op\ WExp \mid n,$$

where $op \in \{+, -, *\}$ and $n \in \mathbb{N}$. We define the set **WEXP** to be the set of all weightexpressions.

A.2 Semantics

$\mathcal{M}odel[\![\texttt{stateset}\ SSet\ \texttt{weightset}\ WSet\ \texttt{in rules}\ ERules; URules]\!] =$
$(\mathcal{SS}et[\![SSet]\!] \times \mathcal{WS}et[\![WSet]\!], \mathcal{ER}ules[\![ERules]\!], \mathcal{UR}ules[\![URules]\!], c_0)$

$\mathcal{SS}et[\![s_1 : S_1; \ldots s_n : S_n]\!] = S_1 \times \cdots \times S_n$

$\mathcal{WS}et[\![w_1 : W_1; \ldots w_n : W_n]\!] = W_1 \times \cdots \times W_n$

$\mathcal{ER}ules[\![<Rule_1>, \ldots, <Rule_n>]\!] = \bigcup_{i=1}^{n} \mathcal{R}ule[\![Rule_i]\!]$

$\mathcal{UR}ules[\![[Rule_1], \ldots, [Rule_n]]\!] = \bigcup_{i=1}^{n} \mathcal{R}ule[\![Rule_i]\!]$

$\mathcal{R}ule[\![Pre- > Upd]\!] = \{(c_1, c_2) \mid c_1 \in \mathcal{P}re[\![Pre]\!] \text{ and } (c_1, c_2) \in \mathcal{U}pd[\![Upd]\!]\}$

$\mathcal{P}re[\![s_1 \in A_1, \ldots, s_n \in A_n]\!] = \{c \mid c(s_i) \in A_i \text{ for all } i = 1 \ldots n\}$

$\mathcal{U}pd[\![s_1 :\in SExp_1; \ldots; s_n :\in SExp_n; w_1 := WExp_1; \ldots; w_k := WExp_k]\!] =$
$\{(c_1, c_2) \mid c_2 = (s_1, \ldots, s_n, w_1, \ldots w_k) \text{ and } s_i \in \mathcal{SE}xp_i[\![SExp_i]\!]c_1 \text{ and } w_j = \mathcal{WE}xp_j[\![WExp_j]\!]c_1$
$\text{for all } i = 1 \ldots n \text{ and } j = 1 \ldots k\}$

$\mathcal{SE}xp_i[\![s_i]\!] = \{(c, a_i) \mid a_i = c(s_i)\}$
$\mathcal{SE}xp_i[\![A_i]\!] = \{(c, a_i) \mid a_i \in A_i \text{ where } A_i \subseteq S_i\}$

$\mathcal{WE}xp[\![w_l]\!]c = c(w_l)$
$\mathcal{WE}xp[\![WExp_1\ op\ WExp_2]\!]c = \mathcal{WE}xp[\![WExp_1]\!]c\ op\ \mathcal{WE}xp[\![WExp_2]\!]c \text{ where } op \in \{+, -, *\}$
$\mathcal{WE}xp[\![n]\!]c = n \text{ where } n \in \mathbb{N}$

Table A.2: Denotational Semantics of LEG.

To give a proper semantic for LEG, we must first define some collections to describe the sets of syntactic expressions. Let **EGAMES** be the collection of all Energy Games. Let **FinSET** be the collection of all finite sets.

LEG defines a two-player game G:

$$G = (C, \rightarrow_\exists, \rightarrow_\forall, c_0)$$

where C is a set of weighted game configurations, $\rightarrow_\exists$ is the existential transitions, $\rightarrow_\forall$ is the universal transitions and c_o is the initial weighted configuration.

The weighted configurations are the Cartesian product of all statesets and all weightsets given

$\mathcal{M}odel$: **MODEL** $\rightarrow$ **EGAMES**
$\mathcal{SS}et$: **SSET** $\rightarrow$ **FinSET**
$\mathcal{WS}et$: $WSET \rightarrow$ **FinSET**
$\mathcal{ER}ules$: **ERULES** $\rightarrow 2^{C\times C}$
$\mathcal{UR}ules$: **URULES** $\rightarrow 2^{C\times C}$
$\mathcal{R}ule$: **RULE** $\rightarrow 2^{C\times C}$
$\mathcal{P}re$: **PRE** $\rightarrow 2^{C}$
$\mathcal{U}pd$: **UPD** $\rightarrow 2^{C\times C}$
$\mathcal{SE}xp_i$: $\mathbf{SEXP_i} \rightarrow 2^{C\times S_i}$
$\mathcal{WE}xp$: **WEXP** $\rightarrow (C \rightarrow \mathbb{Z})$

Table A.3: Signatures of Denotational Semantics of LEG.

by M, i.e.

$$C = S_1 \times \cdots \times S_n \times W_1 \times \cdots \times W_k = \prod_{i=1}^{n} S_i \times \prod_{j=1}^{k} W_j.$$

The existential transitions have the signature $\rightarrow_{\exists} \subseteq C \times C$ and the universal transitions also have the signature $\rightarrow_{\forall} \subseteq C \times C$.

We introduce the semantic function $\mathcal{M}odel$ with the signature **MODEL** $\rightarrow$ **EGAMES**. The function is defined as

$$\mathcal{M}odel[\![\texttt{stateset } SSet \texttt{ weightset } WSet \texttt{ in rules } ERules; URules]\!] =$$

$$(\mathcal{SS}et[\![SSet]\!] \times \mathcal{WS}et[\![WSet]\!], \mathcal{ER}ules[\![ERules]\!], \mathcal{UR}ules[\![URules]\!], c_0).$$

Likewise we introduce the two functions $\mathcal{SS}et$ and $\mathcal{WS}et$ with the signature **SSET** $\rightarrow$ **FinSET** and **WSET** $\rightarrow$ **FinSET**, respectively. We define the semantic functions:

$$\mathcal{SS}et[\![s_1 : S_1; \ldots s_n : S_n]\!] = S_1 \times \cdots \times S_n$$

and

$$\mathcal{WS}et[\![w_1 : W_1; \ldots w_n : W_n]\!] = W_1 \times \cdots \times W_n.$$

We introduce the two semantic functions $\mathcal{ER}ules$ and $\mathcal{UR}ules$, with signatures **ERULES** $\rightarrow$ $2^{C\times C}$ and **URULES** $\rightarrow 2^{C\times C}$. We define these two functions

$$\mathcal{ER}ules[\![<Rule_1>, \ldots, <Rule_n>]\!] = \bigcup_{i=1}^{n} \mathcal{R}ule[\![Rule_i]\!]$$

and

$$\mathcal{UR}ules[\![[Rule_1], \ldots, [Rule_n]]\!] = \bigcup_{i=1}^{n} \mathcal{R}ule[\![Rule_i]\!],$$

respectively.

We define two weighted configurations (c_1, c_2) to be in a rule, $[\![Pre \rightarrow Upd]\!]$, if and only if c_1 is in $[\![Pre]\!]$ and (c_1, c_2) are in $[\![Upd]\!]$, i.e.

$$(c_1, c_2) \in [\![Pre \rightarrow Upd]\!] \stackrel{def}{\Longleftrightarrow} c_1 \in [\![Pre]\!] \wedge (c_1, c_2) \in [\![Upd]\!].$$

A rule has the signature $[\![Pre \rightarrow Upd]\!] \subseteq C \times C$, i.e. its range is a mapping from one weighted configuration to another weighted configuration. The semantic function $\mathcal{R}ule$ has the signature $\mathbf{RULE} \rightarrow 2^{C \times C}$. We define the function

$$\mathcal{R}ule[\![Pre- > Upd]\!] = \{(c_1, c_2) \mid c_1 \in \mathcal{P}re[\![Pre]\!] \; and \; (c_1, c_2) \in \mathcal{U}pd[\![Upd]\!]\}.$$

We introduce a notation for referring to the states and weights a configuration is made of: If a weighted configuration c is made up of states and weights, $c = (a_1, \ldots, a_n, v_1, \ldots, v_k)$, then we denote $c(s_i) = a_i$ and $c(w_j) = v_j$.

$[\![Pre]\!]$ describes the conditions under which a rule applies. These conditions are described by a weighted configuration - a rule is enabled if the current weighted configuration is described by the conditions found in Pre. Thus, $[\![Pre]\!]$ must be a set of weighted configurations, so the signature must be $[\![Pre]\!] \subseteq C$. The set of configurations under which the conditions are met, are the possible combinations of states for each statevariable described in $[\![Pre]\!]$. In order to make these configurations into weighted configurations, it is necessary to combine them with weights. Because LEG does not have conditions related to weights, the weights are simply whatever the weights happen to be in c. We introduce the semantic function $\mathcal{P}re$ which has signature $\mathbf{PRE} \rightarrow 2^C$. We define

$$\mathcal{P}re[\![s_1 \in A_1, \ldots, s_n \in A_n]\!] = \{c \mid c(s_i) \in A_i \; for \; all \; i = 1 \ldots n\}.$$

$[\![Upd]\!]$ is to describe the effect on the system, i.e. a change in state and weight. A change in state and weight is a transition from one weighted configuration to another weighted configuration. The signature is thus $[\![Upd]\!] \subseteq C \times C$. Given a weighted configuration c_1 and an update $[\![Upd]\!]$ it is possible to determine a new weighted configuration c_2 by evaluating the state expressions $SExp_i$ and weight expressions $WExp_j$ given by the syntax.

$$(c_1, c_2) \in [\![s_1 :\in SExp_1; \ldots; s_n \in SExp_n; w_1 := WExp_1; \ldots; w_k := WExp_k]\!]$$

$$\stackrel{def}{\Longleftrightarrow}$$

$$\forall i = 1 \ldots n \,.[\![SExp_i]\!]\,(c_1, c_2\,(s_i)) \wedge \forall j = 1 \ldots k.\, c_2(w_j) = [\![WExp_j]\!]c_1.$$

In order to achieve this we introduce the semantic function $\mathcal{U}pd$ with signature $\mathbf{UPD} \rightarrow 2^{C \times C}$. We define this function:

$$\mathcal{U}pd[\![s_1 :\in SExp_1; \ldots; s_n :\in SExp_n; w_1 := WExp_1; \ldots; w_k := WExp_k]\!] =$$

$\{(c_1, c_2) \mid c_2 = (s_1, \ldots, s_n, w_1, \ldots w_k) \; and \; s_i \in \mathcal{SE}xp_i[\![SExp_i]\!]c_1 \; and \; w_j = \mathcal{WE}xp_j[\![WExp_j]\!]c_1$

for all $i = 1 \ldots n \; and \; j = 1 \ldots k\}$.

A stateexpression for changing the state in a configuration is a mapping between a weighted configuration and a state. Stateexpressions have the signature $[\![SExp_i]\!] \subseteq C \times S_i$. The abstract

syntax requires a definition of the semantics for both $(c, a_i) \in [\![s_i]\!]$ and $(c, a_i) \in [\![A_i]\!]$. For the case $(c, a_i) \in [\![s_i]\!]$, the state expression is referring to the current state of the state variable - this is equivalent to no change, $(c, a_i) \in [\![s_i]\!] \overset{def}{\iff} a_i = c(s_i)$. We introduce a semantic function $\mathcal{SE}xp_i$ with signature $\mathbf{SEXP_i} \to 2^{C \times S_i}$. We define the function:

$$\mathcal{SE}xp_i[\![s_i]\!] = \{(c, a_i) \mid a_i = c(s_i)\}.$$

For the case $(c, a_i) \in [\![A_i]\!]$, the resulting tuple must contain the statevariable a_i in any state from the set of states A_i, where $A_i \subseteq S_i$. Thus, $(c, a_i) \in [\![A_i]\!] \overset{def}{\iff} a_i \in A_i$ define the nondeterministic assignment of the statevariable a_i to any state in $A_i \subseteq S_i$. We extend the semantic function $\mathcal{SE}xp_i$ with a definition for the case $[\![A_i]\!]$:

$$\mathcal{SE}xp_i[\![A_i]\!] = \{(c, a_i) \mid a_i \in A_i\}$$

where $A_i \subseteq S_i$.

A weightexpression is a mapping between a weighted configuration c and a new weight value. This gives the weightexpression the signature: $[\![WExp_j]\!] : C \to \mathbb{Z}$. The semantic function is required to cover the cases $[\![n]\!]c$, $[\![w_l]\!]c$ and $[\![wexp_1\ op\ wexp_2]\!]c$. We introduce the semantic function $\mathcal{WE}xp$ with the signature $\mathbf{WEXP} \to (C \to \mathbb{Z})$. We define how the function acts on the three case:

$$\mathcal{WE}xp[\![w_l]\!]c = c(w_l)$$

$$\mathcal{WE}xp[\![WExp_1\ op\ WExp_2]\!]c = \mathcal{WE}xp[\![WExp_1]\!]c\ op\ \mathcal{WE}xp[\![WExp_2]\!]c$$

where $op \in \{+, -, *\}$, and

$$\mathcal{WE}xp[\![n]\!]c = n$$

where $n \in \mathbb{N}$.

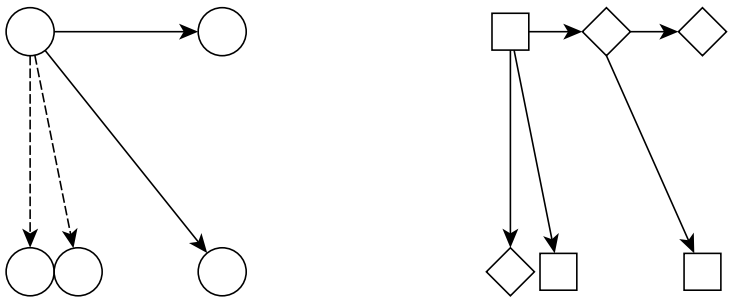
APPENDIX B

TURN-BASED ENERGY GAMES

Earlier sources of energy games present a different way of defining the games. It is possible to express energy games as turn-based models, where each configuration belongs to the existential and universal player respectively. This is for example shown in [13]. In this case, existential locations are marked with a diamond ($\diamond$) and universal locations with a square ($\Box$). We will show that these methods of representing the games are equivalent in expressive power. When we introduce LEG in Chapter 3, we will demonstrate how this language is suited for expressing transitions as existential or universal.

This change means little to the implementation and the semantics of the games. One way of looking at this abstraction is the transformation to the other representation of energy games. When encountering a node which has both existential and universal transitions, the universal player has the choice of playing his own transitions, or surrendering the choice to the existential player. This is reflected in the transformation in Figure B.1b.

We will now show how an energy game represented by owned transitions can be transformed to an energy game with owned states *i.e.* a turn-based energy game. Please refer to the two



(a) $\mathcal{G} = (\mathbf{Q}, \longrightarrow_\exists, \longrightarrow_\forall, q_0)$. (b) $\mathcal{G}_T = (\mathbf{Q}_\exists, \mathbf{Q}_\forall, \longrightarrow, q_0)$.

Figure B.1: Transitions belonging to players, and locations belonging to players.

definitions of energy games found in Figure B.1.

First we must make the transformation from one set of locations $\mathbf{Q}$ to two sets of locations, existential and universal $\mathbf{Q}_\exists, \mathbf{Q}_\forall$.

We denote by G_T the two-player turn-based energy game equivalent of an energy game G, which is obtained from the transformation below.

Theorem B.1 *The turn-based energy game G_T is equivalent in complexity and execution to, and can be transformed to and from an energy game G.*

This is shown in the following transformation:

$$\text{We say that } q \in \mathbf{Q}_\forall \text{ iff } q \longrightarrow_\forall q' \text{ for some } q'.$$
$$\text{And } q \in \mathbf{Q}_\exists \text{ iff } q \longrightarrow_\exists q' \text{ for some } q' \text{ and } q \longrightarrow_\forall q'' \not\in \longrightarrow_\forall \text{ for any } q''.$$

Next we transform the transition relations $\longrightarrow_\exists, \longrightarrow_\forall$ to one transition relation

$$\longrightarrow \subseteq (\mathbf{Q}_\exists \cup \mathbf{Q}_\forall) \times \mathbb{Z}^k \times (\mathbf{Q}_\exists \cup \mathbf{Q}_\forall).$$

For universal transitions we say that

$$\text{if } q \in \mathbf{Q}_\forall \text{ and there exists a transition } q \xrightarrow{\overline{w}}_\forall q' \text{ then } \longrightarrow = \longrightarrow \cup (q, \overline{w}, q').$$

For existential transitions we say that

$$\text{if } q \in \mathbf{Q}_\forall \text{ and } q \xrightarrow{\overline{w}}_\exists q' \text{ then } \longrightarrow = \longrightarrow \cup (q, \overline{0}, q'') \cup (q'', \overline{w}, q')$$
$$\text{where } q'' \in \mathbf{Q}_\exists \text{ is a newly created intermediate existential location.}$$
$$\text{If } q \in \mathbf{Q}_\exists \text{ then every transition is added to } \longrightarrow \text{ without transformation.}$$

Strategies and the definition of them are also defined in the same manner, however, we say that a weighted run for Player 1 ($\exists$) respects a strategy σ if $\sigma((q_0, \overline{v}_0), ..., (q_n, \overline{v}_n)) = (q_{n+1}, \overline{v}_{n+1})$ for all n where $q_n \in \mathbf{Q}_\exists$.

As such we have shown that this definition of energy games can be reduced to a turn-based energy game, and is therefore also subject to the same complexity results.

APPENDIX C

FIXED POINTS

In this chapter we will give some formalization of fixed points, and how they are used to find winning spaces, and thereby winning strategies in energy games. Some of the text in this chapter is based on [3, 17].

Initially, we aim to describe Tarski's Fixed Point Theorem, and then explain how this forms the basis of achieving a fixed point of an energy game. To do so we will first briefly introduce partially ordered sets and complete lattices.

Definition C.1 *Partially Ordered Sets (posets)*

A partially ordered set, or, a poset - is a pair $(\mathbf{D}, \sqsubseteq)$ *where* $\mathbf{D}$ *is a set and* $\sqsubseteq$ *is a binary relation over* $\mathbf{D}$ *for which it holds that:*
$\sqsubseteq$ *is reflexive:* $d \sqsubseteq d$ *for all* $d \in \mathbf{D}$.
$\sqsubseteq$ *is antisymmetric:* $d \sqsubseteq e$ *and* $e \sqsubseteq d$ *implies that* $d = e$ *for all* $d, e \in \mathbf{D}$.
$\sqsubseteq$ *is transitive:* $d \sqsubseteq e \sqsubseteq d'$ *implies that* $d \sqsubseteq d'$ *for all* $d, d', e \in \mathbf{D}$.

We say that a pair $(\mathbf{D}, \sqsubseteq)$ is *totally ordered* if, for all $d, e \in \mathbf{D}$, $d \sqsubseteq e$ or $e \sqsubseteq d$ holds.

Definition C.2 *Lower and Upper Bounds*

If $(\mathbf{D}, \sqsubseteq)$ *is a poset, and* $\mathbf{X}$ *is a set, then consider* $\mathbf{X} \subseteq \mathbf{D}$. *We say that* $d \in \mathbf{D}$ *is an upper bound of* $\mathbf{X}$ *iff* $x \sqsubseteq d$ *for all* $x \in \mathbf{X}$.
d *is the least upper bound (lub) of* $\mathbf{X}$, *written* $\bigsqcup \mathbf{X}$, *iff*
d *is an upper bound for* $\mathbf{X}$ *and*
$d \sqsubseteq d'$ *for every* $d' \in \mathbf{D}$ *that is an upper bound for* $\mathbf{X}$.

We say that $d \in \mathbf{D}$ *is a lower bound for* $\mathbf{X}$ *iff* $d \sqsubseteq x$ *for all* $x \in \mathbf{X}$.
d *is the greatest lower bound (glb) of* $\mathbf{X}$, *written* $\bigsqcap \mathbf{X}$ *iff*
d *is a lower bound for* $\mathbf{X}$ *and*
$d' \sqsubseteq d$ *for every* $d' \in \mathbf{D}$ *that is a lower bound for* $\mathbf{X}$.

Definition C.3 *Complete Lattices*

A poset $(\mathbf{D}, \sqsubseteq)$ *is a complete lattice iff* $\bigsqcap \mathbf{X}$ *and* $\bigsqcup \mathbf{X}$ *exist for every subset* $\mathbf{X}$ *of* $\mathbf{D}$*. A complete lattice has a least element* $\bot = \bigsqcap \mathbf{D}$ *(bottom element), and a greatest (top) element* $\top = \bigsqcup \mathbf{D}$*.*

Monotonic Functions and Fixed Points Let $(\mathbf{D}, \sqsubseteq)$ be a poset. We say that a function $f : \mathbf{D} \to \mathbf{D}$ is monotonic iff $d \sqsubseteq d'$ implies that $f(d) \sqsubseteq f(d')$ for all $d, d' \in \mathbf{D}$. An element $d \in \mathbf{D}$ is called a fixed point of f iff $d = f(d)$.

When computing the fixed point of the configuration space (the winning space), we must therefore define a monotonic function $f : \mathbf{W} \to \mathbf{W}$ to be able to compute it.

Theorem C.1 *Tarski's Fixed Point Theorem*

Let $(\mathbf{D}, \sqsubseteq)$ *be a complete lattice, and* $f : \mathbf{D} \to \mathbf{D}$ *be monotonic. Then* f *has a largest fixed point* z_{max} *and a least fixed point* z_{min} *given by.*

$$z_{max} = \bigsqcup \{d \in \mathbf{D} \mid d \sqsubseteq f(d)\},$$
$$z_{min} = \bigsqcap \{d \in \mathbf{D} \mid f(d) \sqsubseteq d\},$$

A full proof for this theorem is found in [3, 17].

Computation of Greatest Fixed Point Let $(\mathbf{2^W}, \subseteq)$ be our complete lattice and $f : \mathbf{2^W} \to \mathbf{2^W}$ a monotonic function, where $\mathbf{W}$ is the set of all possible weighted configurations.

Then, from Tarski's theorem we know that the greatest (and least) fixed point of f is:

$$z_{max} = f^n(\mathbf{W}) \; z_{min} = f^m(\emptyset) \tag{C.1}$$

Or, in other words, the greatest fixed point is equal to the function of the top element for some $n, m \in \mathbb{N}$, where $\emptyset = \bot$ and $\mathbf{W} = \top$. Since f is monotonic we know that we have the non-growing sequence

$$\mathbf{W} \geq f(\mathbf{W}) \geq f^2(\mathbf{W}) \geq ... \geq f^n(\mathbf{W})$$

and the non-decreasing sequence

$$\emptyset \leq f(\emptyset) \leq f^2(\emptyset) \leq ... \leq f^m(\emptyset)$$

As an example of a fixed point computation, consider the fixpoint computation of the type of energy game given in this report. If we let $\mathbf{W} \in \mathbf{2^W}$ be a set of configurations of a game G, c an element of $\mathbf{W}$, and $f : \mathbf{2^W} \to \mathbf{2^W}$ be defined as:

$$\begin{aligned} f(\mathbf{W}_n) = \{&\mathbf{W}_{n+1} \mid c \text{ is in } \mathbf{W}_{n+1} \text{ iff} \\ &\text{there exists a transition } (c, \overline{w}, c') \in \longrightarrow_\exists \\ &\text{and for every existing transition } (c, \overline{w}, c') \in \longrightarrow_\forall \\ &c' \text{ is in } \mathbf{W}_n\} \end{aligned}$$

Then it should be apparent that f is indeed monotonic, since $\mathbf{W} \geq f(\mathbf{W}) \geq f^2(\mathbf{W}) \geq ... \geq f^n(\mathbf{W})$ and $\emptyset \leq f(\emptyset) \leq f^2(\emptyset) \leq ... \leq f^m(\emptyset)$ and that $(\mathbf{2^W}, \subseteq)$ is a complete lattice.

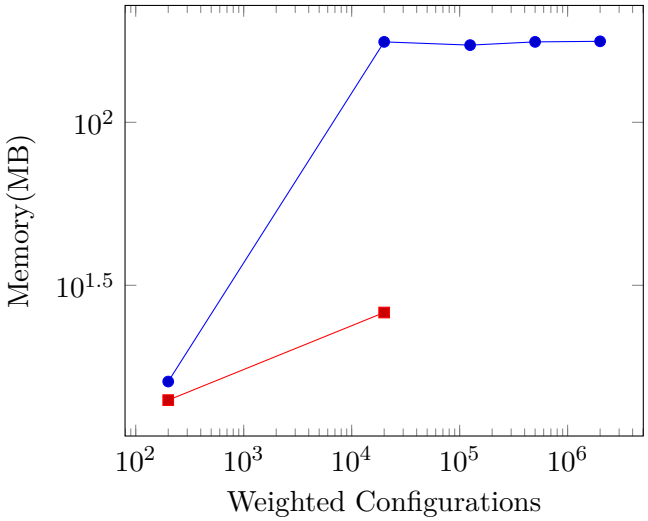
Knowing this, we can derive from the fixed point theorem, that since $\mathbf{2^W}$ is a complete lattice, and f is monotonic, then , by application of Tarski's Fixed Point Theorem, we know that the greatest fixed point can be found by repeated computation of the top element of $\mathbf{2^W}$ in $f^n(\mathbf{W})$(EquationC.1), and a unique greatest fixed point is then guaranteed to be found in a finite number of steps n.

APPENDIX D

RAW TEST RESULTS

Test case	Weighted Configurations	Memory by symbolic (MB)	Memory by explicit (MB)
10s2ss	200	16.02	14.05
100s2ss	20000	176.5	26.09
250s2ss	125000	172.5	*Timeout*
500s2ss	500000	176.5	*Timeout*
1000s2ss	2000000	177.2	*Timeout*

(a) Case **Ms2ss**: Memory.

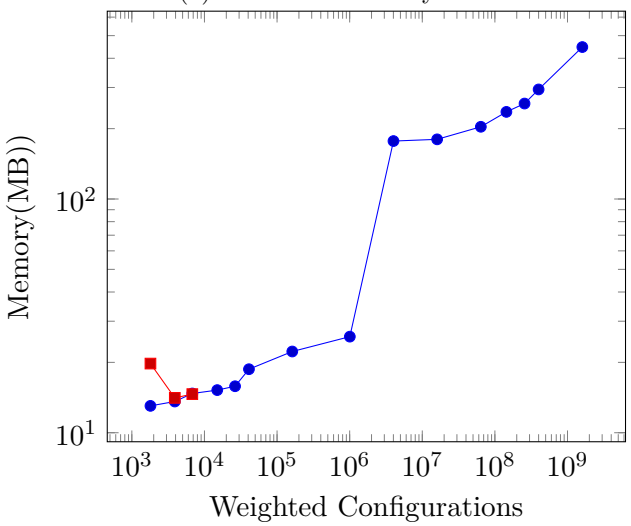


(b) Plot of memory for **Ms2ss**, found in Table D.1a.

Table D.1: Test case **Ms2ss**, memory results.

Test case	Weighted Configurations	Memory by symbolic (MB)	Memory by explicit (MB)
ow10x	1804	13.03	19.77
ow15x	3904	13.6	14.1
ow20x	6804	14.72	14.63
ow30x	15004	15.22	*Timeout*
ow40x	26404	15.81	*Timeout*
ow50x	41004	18.7	*Timeout*
ow100x	162004	22.26	*Timeout*
ow250x	1005004	25.77	*Timeout*
ow500x	4010004	177	*Timeout*
ow1000x	16020004	180	*Timeout*
ow2000x	64040004	203.7	*Timeout*
ow3000x	144060004	235.9	*Timeout*
ow4000x	256080004	256.1	*Timeout*
ow5000x	400100004	294.3	*Timeout*
ow10000x	1600200004	446.8	*Timeout*

(a) Case **ow**: Memory.

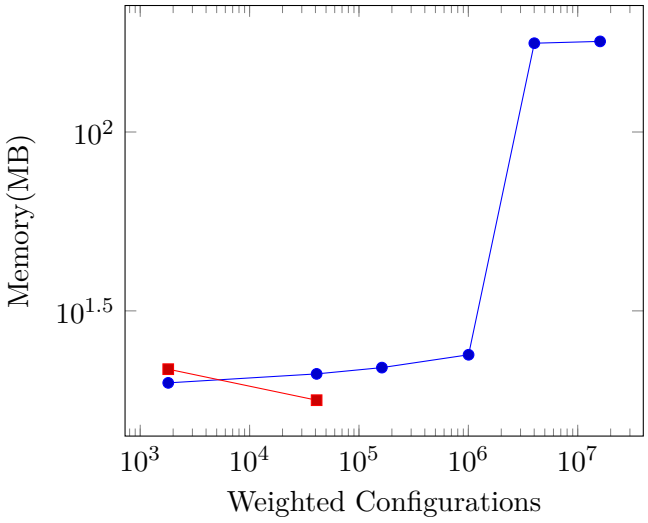


(b) Plot of memory for **ow**, found in Table D.2a.

Table D.2: Test case **ow**, memory results.

Test case	Weighted Configurations	Memory by symbolic (MB)	Memory by explicit (MB)
war10x	1804	19.9	21.73
war50x	41004	21.08	17.8
war100x	162004	21.95	*Timeout*
war250x	1005004	23.84	*Timeout*
war500x	4010004	177.1	*Timeout*
war1000x	16020004	179.2	*Timeout*

(a) Case **war**: Memory.

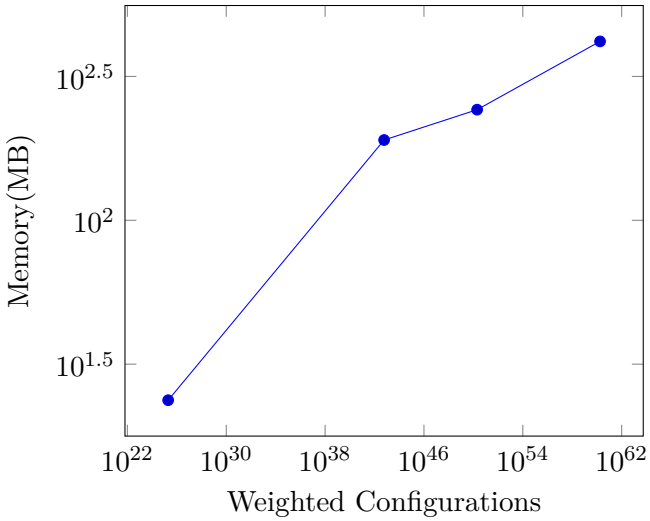


(b) Plot of memory for **war**, found in Table D.3a.

Table D.3: Test case **war**, memory results.

Test case	Weighted Configurations	Memory by symbolic (MB)	Memory by explicit (MB)
10s25ss	2×10^{25}	23.7	*OOM*
50s25ss	5.96046×10^{42}	190.1	*OOM*
100s25ss	2×10^{50}	242.3	*OOM*
250s25ss	1.77636×10^{60}	418.5	*OOM*
500s25ss	5.96046×10^{67}	*OOM*	*OOM*
1000s25ss	2×10^{75}	*OOM*	*OOM*

(a) Case **Ms25ss**: Memory.

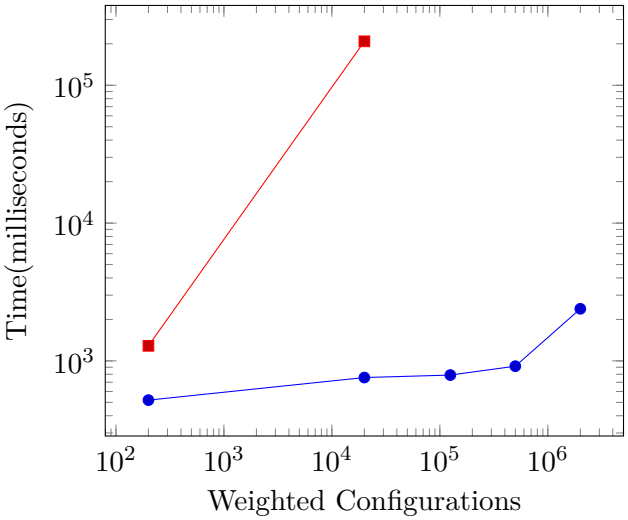


(b) Plot of memory for **Ms25ss**, found in Table D.4a.

Table D.4: Test case **Ms25ss**, memory results.

Test case	Weighted Configurations	Time by symbolic (ms)	Time by explicit (ms)
10s2ss	200	519.2873	1287.193
100s2ss	20000	757.4245	208429.8882
250s2ss	125000	789.4276	*Timeout*
500s2ss	500000	915.4329	*Timeout*
1000s2ss	2000000	2388.4742	*Timeout*

(a) Case **Ms2ss**: Time.

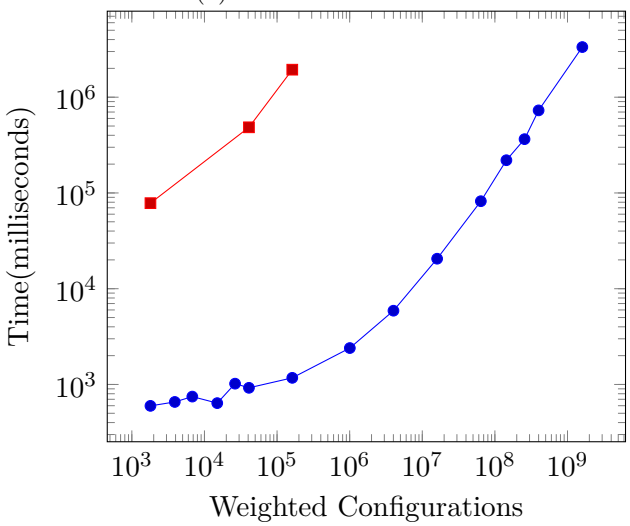


(b) Plot of time for **Ms2ss**, found in Table D.5a.

Table D.5: Test case **Ms2ss**, time results.

Test case	Weighted Configurations	Time by symbolic (ms)	Time by explicit (ms)
ow10x	1804	597.709	78260.4205
ow15x	3904	656.7422	484894.9852
ow20x	6804	746.9507	1938694.073
ow30x	15004	638.447	*Timeout*
ow40x	26404	1019.3067	*Timeout*
ow50x	41004	922.4052	*Timeout*
ow100x	162004	1172.4107	*Timeout*
ow250x	1005004	2400.2739	*Timeout*
ow500x	4010004	5891.5896	*Timeout*
ow1000x	16020004	20568.1978	*Timeout*
ow2000x	64040004	82016.3743	*Timeout*
ow3000x	144060004	219741.7351	*Timeout*
ow4000x	256080004	364607.3954	*Timeout*
ow5000x	400100004	728597.5853	*Timeout*
ow10000x	1600200004	3339701.68	*Timeout*

(a) Case **ow**: Time.

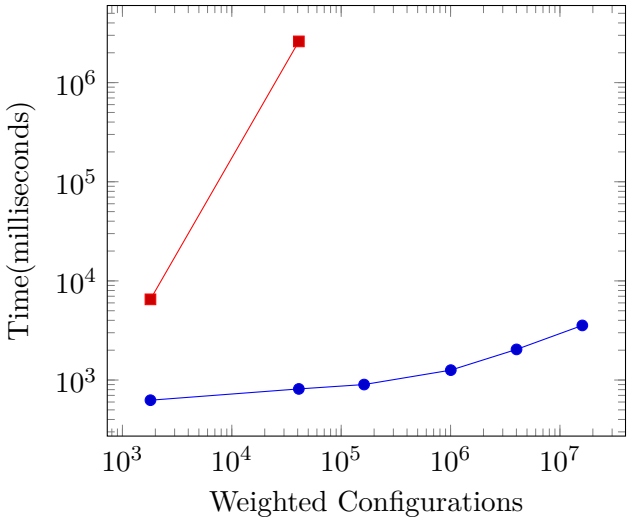


(b) Plot of time for **ow**, found in Table D.6a.

Table D.6: Test case **ow**, time results.

Test case	Weighted Configurations	Time by symbolic (ms)	Time by explicit (ms)
war10x	1804	625.8806	6515.9312
war50x	41004	813.4001	2605022.448
war100x	162004	900.3552	*Timeout*
war250x	1005004	1258.4539	*Timeout*
war500x	4010004	2036.4318	*Timeout*
war1000x	16020004	3555.194	*Timeout*

(a) Case **war**: Time.

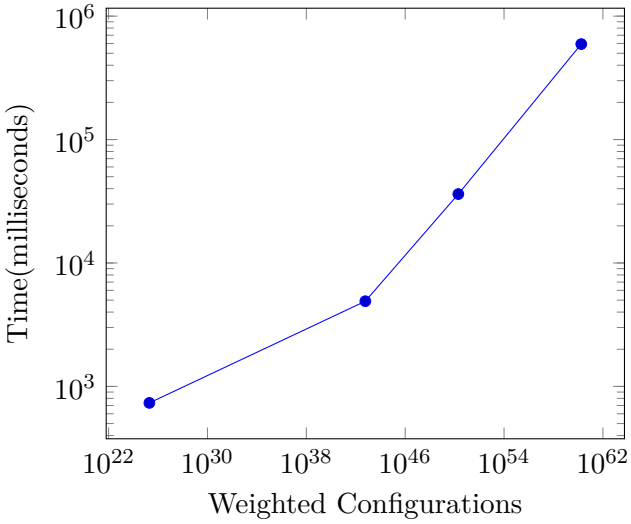


(b) Plot of Time for **war**, found in Table D.7a.

Table D.7: Test case **war**, time results.

Test case	Weighted Configurations	Time by symbolic (ms)	Time by explicit (ms)
10s25ss	2×10^{25}	735.1674	OOM
50s25ss	5.96046×10^{42}	4900.5141	OOM
100s25ss	2×10^{50}	36165.7469	OOM
250s25ss	1.77636×10^{60}	593529.1146	OOM
500s25ss	5.96046×10^{67}	OOM	OOM
1000s25ss	2×10^{75}	OOM	OOM

(a) Case **Ms25ss**: Time.



(b) Plot of time for **Ms25ss**, found in Table D.8a.

Table D.8: Test case **Ms25ss**, time results.