

# A Formalism for Build Order Search in *StarCraft Brood War*

Severin Wyss

December 1, 2016

## **Abstract**

We consider real-time strategy (RTS) games which have temporal and numerical aspects and pose challenges which have to be solved within limited search time. These games are interesting for AI research because they are more complex than board games. Current AI agents cannot consistently defeat average human players, while even the best players make mistakes we think an AI could avoid. In this thesis, we will focus on *StarCraft Brood War*. We will introduce a formal definition of the model Churchill and Buro [2] proposed for *StarCraft*. This allows us to focus on Build Order optimization only. We have implemented a base version of the algorithm Churchill and Buro used for their agent. Using the implementation we are able to find solutions for Build Order Problems in *StarCraft Brood War*.

## **Acknowledgements**

I would like to thank Malte Helmert that I can write my thesis in his research group and for the motivating topic.

Furthermore I like to thank Dave Churchill not only for his work but also for helping me resolve questions about his agent.

Very special thanks go to Martin Wehrle who supported me all the way through with valuable advice, regular meetings and constructive feedback, as well as being available for questions at any time.

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Background</b>	<b>5</b>
2.1	State Space Search . . . . .	6
2.2	Numerical Values . . . . .	7
2.3	Time . . . . .	8
2.4	StarCraft Brood War . . . . .	9
<b>3</b>	<b>Churchill and Buro’s Approach</b>	<b>10</b>
3.1	Simplifications on State Spaces for Build Order Optimization in <i>StarCraft</i> . . . . .	10
3.2	The Fast Forward Mechanism . . . . .	13
3.3	Search Algorithm . . . . .	13
<b>4</b>	<b>State Space Formalization</b>	<b>13</b>
4.1	States . . . . .	14
4.2	Fast Forward . . . . .	18
4.3	Transitions . . . . .	22
4.4	Entry and Solution . . . . .	25
4.5	Example . . . . .	27
<b>5</b>	<b>Experiments</b>	<b>28</b>
<b>6</b>	<b>Conclusion</b>	<b>29</b>
<b>A</b>	<b>Complete List of Orders in our Model</b>	<b>30</b>

# 1 Introduction

Real-time strategy (RTS) games pose interesting challenges for AI research both when programming them and while playing. In this thesis we will only look at playing the game. Like boardgames RTS games are completely static and deterministic. Also normally we only consider the scenario with 2 agents playing the game as adversaries like we do in *Chess* or *Go*. Scenarios with cooperating agents would boil down to a similar task, extended by the need to communicate.

Usually RTS games are significantly more complex than boardgames. The complexity comes from continuous values and only partial observability as well as the way actions influence the game state. In board games actions are instantaneous changes like moving a *pawn* from one field to another. However in RTS games, the player usually issues orders to units. These units then perform a series of jobs like moving or building, as a result of the given order. Though there might be very different games which are classified as RTS, the scenario we are interested in is the one present in most famous RTS games like *StarCraft*, *Age of Empires 2* or *Supreme Commander*.

In these games, generally only the positions of game elements are represented by continuous values, while the rest of the game is discrete. This however suffices for a search space explosion, as already the set of possible “move”-actions for a single unit in a single moment is infinite. Optimizations in game engines typically reduce the actual size of this set to a natural number of options. However, this (at least in modern games) will still be much bigger than the set of 28 “move”-options one *queen* in *chess* would have in the middle of an empty board. Now we cannot compute a set of successors with infinite magnitude for every state, so we cannot solve this task with brute force. Finding abstractions which preserve optimality or finding as good as possible, likely suboptimal solutions are therefore reasonable approaches to this part of the problem.

The next interesting aspect is the time. “Classical” actions do describe an instantaneous change of a state. Although it is possible to model a world which is static and deterministic in time steps, it is very tedious. We will explore the difficulties in such domains and show how temporal action approach them.

Another interesting aspect of RTS games is the partial observability. Human understanding of the scenario dictates, that the gathering of information about the game-state is very important. This assumption fits to the concept of reaction - real time planning on the go - for which this information results in a possibly significant search-space reduction (worst case no difference) which is important in case of limited computation power. In this thesis we will only consider the search for Build Order for one player independently from the opponent. Therefore our problem is completely observable as we only concentrate on game elements which are constantly visible.

RTS games know different kinds of goal states, *StarCraft* requires one player without any building. *Age of Empires 2* is not over until the last Unit or Building from a player is gone and *Supreme Commander* ends as soon as the *ACU*-unit of a Player is killed - similar to the *king* in *chess*. The common characteristic of these games is, that every unit has a certain role towards the

goal of the game. A very naive distinction into 3 types is very common for both units and structures: worker, fighter and utility. For our purpose, units and structures behave the same way. For brevity, we will denote the union of units and structures as units. Simplified a game works as follows: workers collect resources and use these resources to produce units, fighters fight and utility units increase the fighting potential of fighters or can boost workers. Depending on the situation, a unit might take any of these roles or even several at once, this is very much influenced by the game and situation. In *StarCraft*, a unit which normally collects resources and constructs buildings can also engage in combat, taking on the role of a fighter, or repair another unit and therefore be a utility unit. In most games there are also so called Upgrades. Upgrades are global variables influencing the performance of units.

In this thesis, we consider *StarCraft Brood War* because much research about RTS games in the last years has focused on it. This is due to a community hack called *BWAPI* [6] that allows to test implementations directly in the game itself. In addition, this also allows for human versus AI comparisons. There have been annual competitions between universities as well as hobby programmers using this API. While this thesis was written *DeepMind* and *Blizzard* announced [7] that they will publish an API to *StarCraft 2* (the successor of *StarCraft Brood War*) allowing AI tournaments. This research field might gain a lot of popularity in the years to come.

In this thesis, we focus on Build Order search for *StarCraft Brood War*. Colloquially the term Build Order is often used as an expression for the strategy of an agent. We use the term Build Order to describe a series of orders to units, which reaches a given goal. These orders instruct the building of units and researching of upgrades. A goal can be any combination of units, upgrades and resources. RTS games are usually war simulations. These simulations progress in fixed time steps, so called frames. All intervals in the game are fixed to numbers of frames. We use the term make span to denote the duration of a Build Order. Intuitively, the make span is the amount of frames that is required to build and collect the components required by a goal when following the Build Order.

The goal of this thesis is to provide a state space model which accurately represents a simplified *StarCraft Brood War*. To achieve this we make use of the simplifications Churchill and Buro [2] introduced for *StarCraft Brood War*. Our formalism is held closer to the game, while being similarly powerful as the model of Churchill and Buro. Additionally we provide a basic implementation of a planner for Build Order search. For this we use the algorithm proposed by Churchill and Buro and adapt it to our model. Our implementation is able to find correct solutions for Build Order problems.

## 2 Background

In this section we will introduce the formalism we used with special empathise on some topics which go beyond classical state space search. We do not aim

at building a new or even innovative formalism but to show the applicability and correctness of the state-space search we construct later while also pointing out what the author learned about the topic. Our model will use a less formal representation of this formalism.

## 2.1 State Space Search

In state space search we consider an environment which is static, deterministic, fully observable, discrete and there is only a single agent present. As a basis for this thesis, we recapitulate the relevant definitions from the course Foundations of Artificial Intelligence [5]. Our formalism uses SAS<sup>+</sup> extended with conditional effects for states and actions.

### Definition 2.1.1 (State Space)

A state space or transition system is a 6-tuple  $S = \langle S, A, cost, T, s_0, S_* \rangle$  with

- $S$  finite set of states
- $A$  finite set of actions
- $cost : A \rightarrow \mathbb{R}_0^+$  action costs
- $T \subset S \times A \times S$  transition relation; deterministic in  $\langle s, a \rangle$ .
- $s_0 \in S$  initial state
- $S_* \subset S$  set of goal states

### Definition 2.1.2 (Transition determinism)

Let  $S = \langle S, A, cost, T, s_0, S_* \rangle$  be a state space.

The triples  $\langle s, a, s' \rangle \in T$  are called (state) transitions.

We say  $S$  has the transition  $\langle s, a, s' \rangle$  if  $\langle s, a, s' \rangle \in T$ . We write this as  $s \xrightarrow{a} s'$ , or  $s \rightarrow s'$  if  $a$  does not matter.

Transitions are deterministic in  $\langle s, a \rangle$ : it is forbidden to have both  $s \xrightarrow{a} s_1$  and  $s \xrightarrow{a} s_2$  with  $s_1 \neq s_2$ .

### Definition 2.1.3 (Terminology from Graph theory)

Let  $S = \langle S, A, cost, T, s_0, S_* \rangle$  be a state space.

Let  $s, s' \in S$  be states with  $s \rightarrow s'$ .

- $s$  is a predecessor of  $s'$
- $s'$  is a successor of  $s$

If  $s \xrightarrow{a} s'$ , then action  $a$  is applicable in  $s$ .

### Definition 2.1.4 (Path)

Let  $S = \langle S, A, cost, T, s_0, S_* \rangle$  be a state space.

Let  $s^{(0)}, \dots, s^{(n)} \in S$  be states and  $\pi_1, \dots, \pi_n \in A$  be actions such that  $s^{(0)} \xrightarrow{\pi_1} s^{(1)}, \dots, s^{(n-1)} \xrightarrow{\pi_n} s^{(n)}$ .

- $\pi = \langle \pi_1, \dots, \pi_n \rangle$  is a path from  $s^{(0)}$  to  $s^{(n)}$
- length of  $\pi : |\pi| = n$
- cost of  $\pi$ :  $cost(\pi) = \sum_{i=1}^n cost(\pi_i)$

**Definition 2.1.5** (Reachable, Solution, Optimal)

Let  $S = \langle S, A, cost, T, s_0, S_* \rangle$  be a state space.

- state  $s$  is reachable if a path from  $s_0$  to  $s$  exists
- paths from  $s \in S$  to some state  $s_* \in S_*$  are solutions for/from  $s$
- solution for  $s_0$  are called solutions for  $S$
- optimal solutions (for  $s$ ) have minimal costs among all solutions (for  $s$ )

We will use these definitions as they stand for our formalism.

## 2.2 Numerical Values

With only the classical state-space definition it is not possible to describe an RTS game as some variables in RTS games take on continuous values or can take infinitely many different values. Although they are mostly bounded in one or the other way, they still produce a infinite state space. One element which is almost always a continuous value in modern RTS is the position, which is a combination of two or even three floating point numbers - x and y coordinates on the map (and z the hight), generally bounded by the map size but being continuous still having a infinitely great domain.

To be able to handle such variables we have to use numerical values instead of finite domains. Additionally we also need mathematical and logical operations on these numbers.

We will use  $\mathbb{N}$  to represent the natural numbers without zero. Then  $\mathbb{N}_0 := \mathbb{N} \cup \{0\}$ ,  $\mathbb{N}^\infty := \mathbb{N} \cup \{\infty\}$  and  $\mathbb{N}_0^\infty := \mathbb{N} \cup \{\infty, 0\}$ . Analogue we denote  $\mathbb{Z}^{\infty, -\infty} := \mathbb{Z} \cup \{\infty, -\infty\}$  for the integer numbers and  $\mathbb{R}^{\infty, -\infty} := \mathbb{R} \cup \{\infty, -\infty\}$  for the real numbers. As usual  $\mathbb{N}_0^\infty \in \mathbb{Z}^{\infty, -\infty} \in \mathbb{R}^{\infty, -\infty}$  and definitions over  $\mathbb{R}^{\infty, -\infty}$  hold over all other sets.

We use the relation operators  $<, \leq, =, \geq, >, \neq$  as usual. As for the infinity values, we define  $\infty > r$  for all  $r \in \mathbb{R}$ .

We use the mathematical operators  $+, -, \cdot$  as usual over the numbers. We will also define the division  $/$  in a restricted way. Let  $a \in \mathbb{R}, b \in \mathbb{R} \setminus \{0\}$  then  $a/b$  is defined as usual,  $b$  must not be  $0, \infty$  or  $-\infty$ . To ensure this, we will only ever allow division by constant non zero finite numbers.

For the infinity value we define the operators as follows:

Let  $r \in \mathbb{R}$  be a rational number and  $\infty$  be infinity, then

- $\infty + r = \infty$  (therefore  $r - \infty = -\infty$ )
- $\infty - r = \infty$

- $\infty \cdot r = \infty$  iff  $r \neq 0$
- $\infty \cdot r = 0$  iff  $r = 0$
- $\infty/r = \infty$  iff  $r \neq 0$

For a state space with numerical values, we also need an assignment of a number to a variable. Let  $a$  be a variable with domain  $\mathbb{R}^{\infty, -\infty}$  and  $b \in \mathbb{R}^{\infty, -\infty}$  then we define  $a := b$  as the assignment of the value  $b$  to the variable  $a$ .

## 2.3 Time

Introducing time into a state-space seems quite easy, basically we can just add a variable time and modify effects to also take some time if we see fitting. If we allow numerical values we can represent any duration. This would result in a state space  $S_{timed} = \langle S, A, cost, T, s_0, S_* \rangle$  with  $S \subset V_{finitedomain} \times V_{numerical}$  where  $time \in V_{numerical}$  with  $time \in \mathbb{R}$ . The time would advance through the effects of actions  $a_i \in A$  ( $time+ = x_i$ )  $\in eff(a_i)$  where  $x_i \in \mathbb{R}$  is the time the action takes to be done.

However this naive approach is not very powerful. To see this let us consider a scenario where Jimmy is either at home or in the city. We can represent this with simple states  $S_{timed} = loc \times time$  where  $dom(loc) = home, city$  and a single operator  $a_{GoCity0} = \langle home, city \wedge (time+ = 30), 1 \rangle$ . Let us also define  $s_0 = home, 0$ . So far this works fine. But while Jimmy is on the train he might engage into other things like reading the newspaper. For this we now introduce a new boolean variable  $read$ . But what about the action of reading? Jimmy might read the newspaper at either location which would be trivial to represent, but when we consider that he might also read it on the way to the city things get more difficult. We could introduce two more actions  $a_{GoCity1} = \langle home \wedge \neg read, city \wedge (time+ = 30) \wedge read, 1 \rangle$  and  $a_2 = \langle \neg read, read, 0 \rangle$ . Although this works for this simple case it can explode very fast when also introducing calling of up to 100 different people, looking at up to 100 pictures on the phone and chatting with any combination of 3 neighbours on the train especially when he could do many of these activities during his ride (  $100 \times 100 \times 7 \times 2 = 140000$  different  $a_{GoCity}$  actions). Another problem arises in this context, let us say Jimmy reads the newspaper where intensively and therefore needs 40 to read it. The activity of reading the news paper is not finished when the action  $a_{GoCity}$ , which is shorter, is over. But this case also gives some hints towards a possible solution. In the classical case an action is instantaneous, we apply it to a state and receive a new state which is independent from the last action. But in a temporal scenario most activities happen over a period of time and not just in sequence, but also concurrently. It is therefore reasonable to include all on-going activities in a state. We also need to consider multiple effects or conditions. On our example, Jimmy can only start to read the newspaper if he has a newspaper. He can only keep on reading it, when he takes the newspaper with him all the time (  $loc_{newspaper} = loc_{Jimmy}$ ). Additionally there might also be a condition



at the end of the action, maybe Jimmy becomes happy when he, after reading the whole newspaper, understood everything.

We need a formalism that can describe preconditions at the beginning of the action, conditions and effects for the duration of the action and also for the end of it. We will therefore use a typical operator model for temporal numerical planing, e.g. as used by Eyerich, Mattmüller and Röger [3].

**Definition 2.3.1** (Temporal Operator)

A temporal operator is a 7-tupel:

$$o_T = \langle d, \text{pre}_{start}(o_T), \text{pre}_{invar}(o_T), \text{pre}_{end}(o_T), \text{eff}_{start}(o_T), \text{eff}_{invar}(o_T), \text{eff}_{end}(o_T), \text{cost}(o_T) \rangle \quad (1)$$

with :

- $\text{pre}_{start}(o_T), \text{pre}_{invar}(o_T), \text{pre}_{end}(o_T)$  which can be all possible combinations of predicate logical or numerical comparisons over all variables in a state  $s \in S$
- $\text{eff}_{start}(o_T), \text{eff}_{invar}(o_T), \text{eff}_{end}(o_T)$  which can be all possible assignments to finite domain variables combined with all numerical operations and assignments on numerical variables.
- $\text{cost}(o_T) \in \mathbb{R}$  the duration of the action.

One very important aspect of temporal actions is the meaning of their duration. If an action terminates at 5 o'clock, the question is whether the end effect (or condition) will be triggered (or checked) before reaching 5 o'clock or when leaving it. If the operator  $o_1$  has the end condition  $\text{pre}_{end}(o_1) = a$  and the operator  $o_2$  has the start effect  $\text{eff}_{start}(o_2) = \neg a$ , the order in which effects are applied and conditions are checked can drastically change the state space. One convention tackling this problem is called the “no moving target”-rule introduced by Fox, Maria and Long, Derek [4]. This conventions introduces restrictions on variables s.t. a variable can only ever be affected by one action at a time and if it is affected, it may not be part of a precondition at the same time.

Because we are building a model for RTS games and are only considering the Build Order part of the game, we do not need to restrict ourself in the way of the “no moving target”-rule. For Build Order search in the simplified model by Churchill and Buro, we can define all temporal operators to terminate right before the frame. In our model, we will use temporal operators implicitly represented by tasks.

## 2.4 StarCraft Brood War

*StarCraft* (published in March, 1998) and its Expansion *StarCraft Brood War* (published in November, 1998) are RTS games published by *Blizzard Entertainment* which take place in a science fiction universe. It is a military simulation

where a player takes control over one of three different races: *Terran* (humanity), *Protoss* (highly developed and telepathic alien race) and *Zerg* (aggressive insectoid lifeforms). The game became the biggest competitive RTS game until its successor *StarCraft 2* took over with its release in 2007.

We will introduce a few game elements from StarCraft. There are some elements that are controlled by the player, those are separated into resources, units, structures and upgrades. As mentioned in the introduction, we use the term units denoting both structures and units. Also we refer to upgrades as researches. The player can give orders to all their units. Units can perform various tasks like moving, attacking or building new units. There are two types of resources, *minerals* and *vespine gas*. For brevity, we will often use *gas* to denote *vespine gas*. Each unit and upgrade requires a certain amount of those two resources to be purchased. Upgrades are researched each by a unit and units are build by other units.

In the competitive setting two player will play against each other on one of a limited number of maps which are usually known beforehand. They start with a small set of units and resources. Through the course of the game they then try to defeat the other player by destroying all structures. Usually both players will start by collecting resources and building more units. When they feel that they have a combination of units which can defeat the other player, they attack. We will not go into more details on combat or movement as we do not consider those aspects in this thesis.

### 3 Churchill and Buro's Approach

In this section, we recapitulate the approach by Churchill and Buro [2]. We discuss the simplifications proposed by them for Build Order optimization, and show their solution algorithm.

#### 3.1 Simplifications on State Spaces for Build Order Optimization in *StarCraft*

- **Combat:** Everything combat related (health, armour, attack) can be omitted as the task does not include the consideration of such. In reality this would not suffice for a good AI. However it is not part of this assignment to construct a good AI for StarCraft, but to build a component which is capable of finding Build Orders which reach a set goal state in as little frames as possible.
- **Cancel:** In *StarCraft* every construction can be cancelled and a portion of the price is returned. For humans this gives the option to cancel something that was not planned to be done, of course an AI will not need this. But besides errors this can play a vital role in high level strategies as this is a game with incomplete information. Letting the opposing player see a building in construction can suggest a strategy while the actual plan is hidden somewhere else or only started after vision is denied again.

However we will ignore it here as every cancel always loses some resources and we are only trying to reduce the make span of our Build Order and not to foul the enemy.

This decision will cost us just a small bit of optimality, there is at least one possible way of using the cancel option which could reduce make span. By starting a building with *Zerg*, we are freeing up 1 supply as a worker unit is morphed into the building. This will allow to build a new unit with that 1 Supply we now have. Cancelling the building afterwards will give us back the drone, thereby resulting in a trade of a small amount of *minerals(+gas)* against 1 supply. We are then in a state where we actually use more supply than we have. This could of course be done as often as we have drones. A very unrealistic case would be if we had all 200 maximum supply filled with drones, 200 *Hatcheries* (*Zerg* equivalent to *Command Center* - produces drones) and 350000 minerals, we could start the construction of 200 new hatcheries which would take 1800 frames to finish and then build 5 times 200 new *Drones* (which take 300 frames plus a few more to get to a position where they can build a *Hatchery*) which each would start the construction of a new *Hatchery* again until the last wave of *Drones* is finished and then cancel all *Hatcheries* in construction, resulting in 1000 *Drones*. A goal which we could not reach in our abstraction. But such a goal is not realistic and in an actual game the usability of this mechanic is not very strong.

- Position: We will ignore the placement of buildings, as it introduces a giant variety to the search space with very little differences between the state which build the structures but moved by a few tiles. It is quite reasonable to assume that there is enough place for any building somewhere near the mineral line where our workers are collecting minerals. A bad player might accidentally build in a way that his *SCVs* cannot get to a free position any more or that there is literally no space left on the map where a big building like a *Command Center* would fit in. However as we are construction a plan, we will know what buildings we are planning to build and therefore it will be relatively easy to place our buildings in a way that avoids this problem.

Of course this will lose us some optimality, as building the structures in a way so that the *SCVs* take as little time as possible travelling would reduce the make span. Also it is often useful to let a *SCV* build several buildings in sequence instead of using two *SCV*'s to build them in parallel, for example building 10 *Supply Depots*, in most realistic scenario a player will not be able to fill 50 supply in the time which is needed to construct a *Supply Depot*, so having 5 *SCV*'s building each two means that we have less time spent travelling.

Building placement can also be decisive in combat by blocking melee units from reaching range units. There are even some strategies where a building is deliberately build near the enemy base, so that the combat units reach

the enemy quicker after they finished producing.

Additionally the resource gathering has also be replaced by a average per worker which is not performing another task. This also removes the option of stopping gathering form the search space, as idle time will always produce a alternative Build Order which is likely worse, but never better.

There is often room of improving the AI behaviour in resource gathering, as only 1 worker can gather from any resource field at a time and the distance between fields and the drop-off-place varies. Instead of trying to improve this, we assume an average constant income rate. For the Build Order optimization we now simply multiply our workers who have no other orders with the average income.

When considering both the building placement and the resource collection we would be able to achieve a shorter make span. But ignoring them drastically reduces the state space. Let us examine this at the very simple goal : build 1 *Supply Depot*. For simplicity let us assume we only have 20 possible location for building it. We start with 4 *SCVs* and 50 *minerals*, so we have to collect 50 *minerals* more which requires 7 mining trips from a *SCV*. We will not consider any action that would not help the goal, so let us say we send all *SCVs* to mining (4 actions) and 3 of them back again when they are finished (3), while the last *SCV* goes to any of those 20 locations that he can reach in the time the other *SCVs* need for their trip (let us assume that are 10, therefore we have 10 more actions). As soon as the *Supply Depot* finishes building we reached our goal with 8 actions, there are 10 equally good solution paths and 10 with have the same amount of actions but take longer. Compared to the search in the abstraction: build a *Supply Depot* is 1 action, no branching. Constructing 2 *Command Centers* takes 800 *minerals*. So we have 102 actions for a already simplified version of *StarCraft* versus the 2 actions in Churchill and Buro's model. The more precise solution might have a make span which is some frames or even seconds faster, however for all but trivial goals it will very likely take longer to find the optimal solution than the make span of the plan and therefore we cannot afford to calculate this online. Given the these simplifications the position of all elements is no longer of any importance for the Build Order.

- Resources: A final simplification introduced by Churchill and Buro is to ignore the limitation of resources. As this tool does not consider combat, it is reasonable to assume that the Build Order should terminate before we enter combat and this is generally before the resources run dry. The speed-up here is of course significant as resource collection does not require any actions.

However this is a dangerous simplification which even in *StarCraft Brood War* might not hold, but would not be tolerable in *StarCraft 2 Legacy of the Void* where resource sources are consumed faster, not even speaking of apply the concept to RTS games in general. There is often a restriction

on how many worker can collect from one resource source in an RTS, in *StarCraft* it is one worker per source, effectively this means that than 4 or more *SCVs* on a mineral patch will not give any more income than with only 3. Already those 3 collect a little slower each than a single one does. When there are 8 mineral patches in a base, only 24 worker do actually increase income, all others would have to go to another base to produce more income.

With these abstraction we can reduce the state space and therefore the computational effort significantly, however we can not tell how near we are to the optimum. In this reduced state space we will be able to check a lot over very different paths, therefore our solution will have a certain quality, being at least close to a local minima which is better than most solutions. Considering that we have limited time for our search this is one of the best ways to take on the challenge.

### 3.2 The Fast Forward Mechanism

Instead of using a time increment action (only advances time) which potentially has to be chosen thousands of times for a single path, just jump forward in time exactly so far that the preconditions of an action are fulfilled. The preconditions of the (fast forward + temporal action) action are, that the preconditions would become available eventually when the time increment action would exist and be chosen continuously.

### 3.3 Search Algorithm

Churchill and Buro motivated a DFS algorithm because it could be terminated at any point and return the best solution found so far. The only weakness here would be that the algorithm explores a path which will not reach a goal state. To avoid this a upper bound is used. This bound is the make span of a trivial plan at the beginning and will be updated when a faster solution is found, thereby always containing the make span of the best currently known solution. The following Algorithm comes from Churchill and Buro.

The algorithm is bounded in time by  $t$ . It is also bounded in make span by  $b$ . Every time a solution is found, it is compared against the current best solution and the one with the shorter make span is kept. The bound will be updated to the shortest make span. The bound is compared against the path cost so far plus a heuristic value. The heuristic gives a naive estimate for resources or units required to reach the goal. For more details about the heuristic refer to the paper of Churchill and Buro [2] page 17.

## 4 State Space Formalization

In this section, we construct a state space for *StarCraft* using the formalisms of Section 3. But we will not represent the whole complexity of the game. Instead

---

**Algorithm 1** Depth First Search Branch & Bound

---

```
1: procedure DFBB( $S$ )
2:   if TimeElapsed  $\geq$   $t$  then return
3:   end if
4:   if  $S$  satisfies  $G$  then
5:      $b \leftarrow \min(b, S_t)$ 
6:     best solution  $\leftarrow$  solutionPath( $S$ )
7:   else
8:     while  $s$  has more children do
9:        $S' \leftarrow S$ .nextChild
10:       $S'$ .parent  $\leftarrow S$ 
11:       $h \leftarrow \text{eval}(S)$ 
12:      if  $S'_t + h < b$  then
13:        DFBB( $S'$ )
14:      end if
15:    end while
16:   end if
17: end procedure
```

---

we will use a set of simplifications proposed by Churchill and Buro [2].

## 4.1 States

We are now ready to formally define the state space for Build Order search. To formally define states, we first need some more definitions.

### Definition 4.1.1 (UnitType)

A UnitType is a label that represents the type for a unit in the game. We distinguish the following UnitType:

- *Terran\_SCV* (representing a worker unit called SCV, creates structures and collects resources)
- *Terran\_Command\_Center* (representing a structure called Command Center, can create SCV's)
- *Terran\_Refinery* (representing a structure called Refinery, used for gathering gas)
- *Terran\_Engineering\_Bay* (representing a structure called Engineering Bay, can research Researches)
- *Terran\_Supply\_Depot* (representing a structure called Supply Depot, produces supply)
- *Terran\_Comsat\_Station* (representing a structure called Comsat Station, is build onto an Command Center)

The set of all *UnitType* labels is denoted as *UnitTypes*. We denote the number of units, i.e.  $|UnitTypes|$ , as  $c_U$ .

We remark that in common RTS games, there are more types of units than mentioned in the above definition. We mention the types in Def. 4.1.1 explicitly because they will be needed later for our further definitions and examples.

Besides units there are also upgrades which may be part of a goal. We will refer to upgrades as researches. This serves to better distinguish them from units.

**Definition 4.1.2** (*ResearchType*)

A *ResearchType* is a label that represents the type for a research in the game. We distinguish the following *ResearchType*:

- *Terran\_Infantry\_Weapons\_1* (Research which increases the damage of infantry units)
- *Terran\_Infantry\_Armor\_1* (Research which increases the armor of infantry units)

The set of all *ResearchType* labels is denoted as *ResearchTypes*. We denote the number of Research label, i.e.,  $|ResearchTypes|$ , as  $c_R$ .

As with the *UnitType* labels, there are typically more *ResearchType* labels than we mention here. But again, we focus on the relevant ones needed for the thesis.

In the following we denote the special labels for “doing nothing” with *IDLE*, and for “gathering gas” with *GasGathering*.

**Definition 4.1.3** (*Task*)

We define a *Task* as a 2-tuple  $t := (w, d)$  where  $w \in UnitTypes \cup ResearchTypes \cup \{IDLE, GasGathering\}$  and  $d \in \mathbb{N}_0^\infty$ . The set of all possible *Task* tuples is denoted as *Tasks*.

The first element ( $w$ ) represents the job that needs to be accomplished. For most *UnitType* labels, this is either producing a *UnitType*, or researching a *ResearchType*, or being *IDLE*. The exception is the worker (*Terran\_SCV*), which can also gather gas. Additionally, by slightly abusing notation and for brevity, a worker who is *IDLE* represents the worker having the job of *GasGathering*. The second element of a *Task* tells how many frames are needed until the job is finished. In this context, the infinity value indicates that the task will carry on forever or until changed.

Based on these definitions, we can now define our representation of units.

In *StarCraft* there are some units, called *Addon*, which are build onto another unit. The two units become partner and can then sometimes produce additional *UnitType*. We represent this concept by a *UnitType* label for each unit. We denote a label for “having no partner” with *NOPARTNER*.

**Definition 4.1.4** (Unit)

We define a Unit as a 5-tuple  $u := (a, b, t, e, n)$  where  $a \in UnitTypes, b \in UnitTypes \cup \{NOPARTNER\}, t \in Tasks, e \in \mathbb{N}_0^\infty, n \in \mathbb{N}_0$ .

We also denote the set of all Unit tuples by Units.

For a Unit  $u$ , the first entry ( $a$ ) represents its type. The second entry represents the *UnitType* of  $u$ 's partner unit, or specifies that no such partner exists. The third entry represents the job that the unit is currently assigned to. The fourth entry is a natural number representing a reservation. After  $e$  frames, the unit will be assigned to a predefined Task (In our representation this is only used for gas gathering). However before that it, besides being IDLE, may perform any Task that will terminate before or exactly when  $e$  reaches zero. Here  $\infty$  represents that  $u$  has no such restriction from this element. The natural number  $n$  represents how many instances exist at the same time. Within the simplification those instances do not need to be differentiated. Therefore we can just count them.

As a small example, consider Unit  $u$  with

$$u = (Terran\_SCV, NOPARTNER, (GasGathering, \infty), \infty, 2).$$

This example denotes that there are two Unit instances of type *Terran\\_SCV* which have no partner assigned, are gathering gas infinitely and both Unit instances have no reservation.

**Definition 4.1.5** (Research)

We define a Research as a 2-tuple  $r := (v, b)$  where  $v \in ResearchTypes, b \in \{true, false\}$ . We denote the set of all Research tuples as Researches.

Furthermore we define the set  $Res := \{(v_1, b_1), \dots, (v_k, b_k) \mid v_i \in ReserachTypes, b_i \in \{true, false\}, v_i \neq v_j \text{ for all } i \neq j, k = |ReserachTypes|\}$

The Boolean value represents, whether the Research has been researched (*true*) or not (*false*). The *Res* definition refers to the set that contains tuples consisting of pairs for each *ResearchType* and Boolean value. Each Research exists at every point in the game, but it is only active after it has been researched. For brevity, we will shortly represent a tuple in *Res* by the set of *ResearchTypes* which are associated with a corresponding true value. This is enough information to distinguish the members of *Res*.

Based on these notions, we define a state as follows.

**Definition 4.1.6** (State)

A State is a 5-tuple  $s := (f, U, R, m, g)$  where  $f \in \mathbb{N}_0, U \subseteq Units, R \in Res, m, g \in \mathbb{R}$

Let States denote the set of all State tuples.

The element  $f$  represents in which frame the game currently is. The element  $U$  represents the units which exist in the current frame. The element  $R$  denotes which Researches are active. The element  $m$  represents the currently available minerals. Analogue  $g$  represents the currently available gas.



As a small example consider the *State*:

$$i_0 := (0, \{(Terran\_SCV, NOPARTNER, (IDLE, \infty), \infty, 4), \\ (Terran\_Command\_Center, NOPARTNER, (IDLE, \infty), \infty, 1), \}, \\ , \{ \}, 50.0, 0.0)$$

The *State*  $i_0$  represents the situation where the game has just begun ( $f = 0$ ). There exist two types of units, one *Command Center* which is *IDLE* and four *SCVs* which are gathering *minerals*. All the *Researches* are inactive. Finally there are 50.0 *minerals* and 0.0 *gas* available.

The supply is a limiting aspect in unit production. To be able to represent this aspect in our model, we introduce a few formulas.

**Definition 4.1.7** (Supply)

A *SupplyConsumption* is an integer representing how much supply a Unit of the respective *UnitType* consumes or produces. We have the following *SupplyConsumption*:

- $c_{Terran\_SCV} := -2$
- $c_{Terran\_Command\_Center} := 20$
- $c_{Terran\_Refinery} := 0$
- $c_{Terran\_Engineering\_Bay} := 0$
- $c_{Terran\_Supply\_Depot} := 16$
- $c_{Terran\_Comsat\_Station} := 0$

Let  $u = (a, b, t, e, n)$  be a Unit  $c_a$  be a *SupplyConsumption*.

We define an injective function *Supply* : *UnitTypes*  $\rightarrow \mathbb{Z}$  that maps *UnitTypes* to corresponding *SupplyConsumption*.

Using this we can define another function *SupplyUnit* : *Units*  $\rightarrow \mathbb{Z}$  for a Unit  $u = (a, b, t, e, n)$  as  $SupplyUnit(u) := Supply(a)$ .

Let  $s = (f, U, R, m, g)$  be a *State*.

We define a third function on *States* as *SupplyState* : *States*  $\rightarrow \mathbb{Z}$  as  $SupplyState(s) := \sum_{u \in U} SupplyUnit(u)$

We have for example the *SupplyConsumption*  $c_{Terran\_SCV} = -2$ , this means a *Terran\\_SCV* uses up 2 *supply*. Meanwhile a *Terran\\_Supply\\_Depot* produces 16 *supply*:  $c_{Terran\_Supply\_Depot} = 16$ . To be able to produce a new *Unit* instance, the total produced *supply* must be bigger or equal to the consumed *supply* plus the *supply* the *UnitType* which we want to construct would consume. In *StarCraft* the *SupplyConsumption* is actually only half of that which we use here. The multiplication with 2 has been proposed in the *BWAPI* [6]. The reason is that there is one unit in *StarCraft Brood War* which requires 0.5 *Supply*. The doubling allows the use of an integer instead of a floating point number.

We also need to define which *UnitType* can be partners with each other.

**Definition 4.1.8** (Partner)

A *PartnerRelation* is a set  $p := \{a, b\}$  where  $a, b \in \text{UnitTypes}$  representing which *UnitType*  $a$  can have which *UnitType*  $b$  as partner and vice versa. We denote the set of *PartnerRelation* sets as *Partners*.

In the definition that we consider here, *Partners* boils down to the simple set  $\text{Partners} := \{\{ \text{Terran\_Command\_Center}, \text{Terran\_Comsat\_Station} \}\}$ . However for general RTS games this set could contain any number of entries.

We define a function  $\text{partner} : \text{UnitTypes} \times \text{UnitTypes} \rightarrow \{\text{true}, \text{false}\}$  as follows:

$$\text{partner}(a, b) := \begin{cases} \text{true}, & \text{if } \{a, b\} \in \text{Partners} \\ \text{false}, & \text{otherwise} \end{cases}$$

A *UnitType* can only have another *UnitType* as partner, if they both are contained in the same *PartnerRelation*.

## 4.2 Fast Forward

In the following we will formalize the a Fast Forward mechanism. For this purpose we first define the components of the *State* after applying the Fast Forward mechanism in relation to the *State*  $s_0$  before the Fast Forward mechanism. In the following definitions, we assume  $s_0 := (f_0, U_0, R_0, m_0, g_0)$  to be a *State* and  $t_{\text{ff}} \in \mathbb{N}_0$  to be a natural number (including zero) which indicates the time by which  $s$  is fast forwarded.

We split the formal definition of the Fast Forward mechanism into several (smaller) definitions, and provide the overall definition of Fast Forward mechanism at the end of the section. We start with the simple definition of advancing time.

**Definition 4.2.1** (Fast Forward Time)

Let  $s_0 := (f_0, U_0, R_0, m_0, g_0)$  be a *State* and  $t_{\text{ff}} \in \mathbb{N}_0$  be a natural number including zero.

The *Fast Forward Time* mechanism  $FF^{\text{time}}$  advances the time of  $s_0$  by  $t_{\text{ff}}$  frames:

$$FF^{\text{time}}(s_0, t_{\text{ff}}) := f_0 + t_{\text{ff}}$$

With  $FF^{\text{time}}$  we update the time. What happens during this time jump is part of the other Fast Forward mechanisms.

As outlined in the section on simplifications, the average income per worker and frame for both minerals  $c_{M/(W*F)}$  and gas  $c_{G/(W*F)}$  is set to a fixed value. We set  $c_{M/(W*F)} := 0.045$  and  $c_{G/(W*F)} := 0.07$  (values empirically determined by Churchill and Buro [2]). For the *Terran* race, the only *UnitType* that classifies as worker is *Terran\_SCV*.

We will now use these average incomes, to define the resource income for our model.

**Definition 4.2.2** (Fast Forward Minerals)

Let  $s_0 := (f_0, U_0, R_0, m_0, g_0)$  be a State and  $t_{ff} \in \mathbb{N}_0$  be a natural number including zero.

Let  $U_{SCV\_IDLE} \subseteq U_0$  be the set of Unit tuples with  $U_{SCV\_IDLE} := \{(a, b, (w, d), e, n) \mid (a = Terran\_SCV, w = IDLE, d = \infty, e > t_{ff})\}$ . Let  $(a_1, b_1(w_1, d_1), e_1, n_1), \dots, (a_l, b_l(w_l, d_l), e_l, n_l)$  where  $l = |U_{SCV\_IDLE}|$  be the Unit tuples in  $U_{SCV\_IDLE}$ . Then let

$$n_{SCV\_IDLE} := \sum_{i=1}^{|U_{SCV\_IDLE}|} (n_i)$$

be the total number of Unit instances in  $U_{SCV\_IDLE}$ .

Let  $U_{SCV\_possible} \subseteq U_0 \setminus U_{SCV\_IDLE}$  be the set of Unit tuples with  $U_{SCV\_possible} := \{(a, b, (w, d), e, n) \mid (a = Terran\_SCV \text{ or } w = Terran\_SCV) \text{ and } e > t_{ff}\}$ . Let  $(a_1, b_1(w_1, d_1), e_1, n_1), \dots, (a_l, b_l(w_l, d_l), e_l, n_l)$  where  $l = |U_{SCV\_possible}|$  be the Unit tuples in  $U_{SCV\_possible}$ . Then let

$$m_{SCV\_possible} := c_{M/(F*W)} \cdot \sum_{i=1}^{|U_{SCV\_possible}|} (n_i \cdot (t_{ff} - d_i))$$

be the total amount of minerals gathered by all Unit instances in  $U_{SCV\_possible}$  during the time progression by  $t_{ff}$ .

Let  $U_{SCV\_reservation} \subseteq U_0 \setminus U_{SCV\_IDLE}$  be the set of Unit tuples with  $U_{SCV\_reservation} := \{(a, b, (w, d), e, n) \mid (a = Terran\_SCV \text{ or } w = Terran\_SCV) \text{ and } e \leq t_{ff}\}$ . Let  $(a_1, b_1(w_1, d_1), e_1, n_1), \dots, (a_l, b_l(w_l, d_l), e_l, n_l)$  where  $l = |U_{SCV\_reservation}|$  be the Unit tuples in  $U_{SCV\_reservation}$ . Then let

$$m_{SCV\_reservation} := \sum_{i=1}^{|U_{SCV\_reservation}|} (n_i \cdot (t_{ff} - d_i - e_i))$$

be the total amount of minerals gathered by all Unit instances in  $U_{SCV\_reservation}$  during the time progression by  $t_{ff}$ .

The Fast Forward Mineral mechanism  $FF^{mineral}$  calculates the total increase of minerals during the next  $t_{ff}$  frames from the current state ( $s_0$ ):  $FF^{mineral}(s_0, t_{ff}) := m + c_{M/(F*W)} \cdot (n_{SCV\_IDLE} \cdot t_{ff}) + m_{SCV\_possible} + m_{SCV\_reservation}$ .

The amount of *minerals* in the State after applying the Fast Forward mechanism is equal to the sum of the current amount plus the average income per frame times the sum of all time a *Unit* instances of *UnitType Terran\\_SCV* will spend *IDLE* during  $t_{ff}$ .

**Definition 4.2.3** (Fast Forward Gas)

Let  $s_0 := (f_0, U_0, R_0, m_0, g_0)$  be a State and  $t_{ff} \in \mathbb{N}_0$  be a natural number including zero.

Let  $U_{SCV\_GG} \subseteq U_0$  be the set of Unit tuples with  $U_{SCV\_GG} := \{(a, b, (w, d), e, n) \mid a = Terran\_SCV, w = GasGathering, d = \infty\}$ . Let  $(a_1, b_1(w_1, d_1), e_1, n_1), \dots, (a_l, b_l(w_l, d_l), e_l, n_l)$  where  $l = |U_{SCV\_GG}|$  be the Unit tuples in  $U_{SCV\_GG}$ .

Then let

$$n_{SCV\_GG} := \sum_{i=1}^{|U_{SCV\_GG}|} n_i$$

be the total number of Unit instances in  $U_{SCV\_GG}$ .

Let  $U_{SCV\_reservation} \subseteq U_0 \setminus U_{SCV\_GG}$  be the set of Unit tuples with  $U_{SCV\_reservation} := \{(a, b, (w, d), e, n) \mid a = \text{Terran\_SCV}, e \leq t_{ff}\}$ . Let  $(a_1, b_1(w_1, d_1), e_1, n_1), \dots, (a_l, b_l(w_l, d_l), e_l, n_l)$  where  $l = |U_{SCV\_reservation}|$  be the Unit tuples in  $U_{SCV\_reservation}$ . Then let

$$g_{SCV\_reservation} := c_{G/(F*W)} \cdot \sum_{i=1}^{|U_{SCV\_possible}|} (n_i \cdot (t_{ff} - e_i))$$

be the total amount of gas gathered by all Unit instances in  $U_{SCV\_reservation}$  during  $t_{ff}$ .

The Fast Forward Gas mechanism  $FF^{gas}$  calculates the total increase of gas during the next  $t_{ff}$  frames from the current State ( $s_0$ ):  $FF^{gas}(s_0, t_{ff}) := g_0 + c_{G/(F*W)} \cdot (n_{SCV\_GG} * t_{ff}) + g_{SCV\_reservation}$ .

The amount of gas in the state after applying the Fast Forward Gas mechanism is equal to the sum of the current amount plus the average income per frame times the sum of all time a Unit instances will spend GasGathering during  $t_{ff}$ .

**Definition 4.2.4** (Fast Forward Research)

Let  $s_0 := (f_0, U_0, R_0, m_0, g_0)$  be a state and  $t_{ff} \in \mathbb{N}_0$  be a natural number including zero. Therefore  $R_0$  denotes the tuple  $((v_1, r_1), \dots, (v_l, r_l))$  with  $l = |R_0|$ .

Let  $R_{ff} \in Res$  be the tuple of Researches  $(v_{i_{ff}}, b_{i_{ff}})$  with  $i = 1, \dots, l$ . Therefore  $|R_{ff}| = |R_0|$  and  $v_i = v_{i_{ff}}$  for all  $i = 1, \dots, l$ .

For all  $r = (v_i, r_i) \in R_0$  the corresponding Research  $(v_{i_{ff}}, b_{i_{ff}}) \in R_{ff}$  is define as follows:

- If there exists a Unit tuple  $((a, b, (w, d), e, n) \in U$  where  $w = v_i$  and  $d \leq t_{ff}$ ) then  $b_{i_{ff}} := true$
- If  $b_i = true$  then  $b_{i_{ff}} := true$
- Else  $b_{i_{ff}} := false$

The Fast Forward Research mechanism  $FF^{research}$  activates the ReserachType which are researched within the next  $t_{ff}$  frames from the current state ( $s_0$ ):  $FF^{research}(s_0, t_{ff}) := R_{ff}$ .

The Research which is build with the Fast Forward Research mechanism is equal to  $R_0$  except, for every Task in  $(w_i, d_i) \in U_0$  with  $d_i \leq t_{ff}$ , where  $w_i \in Researches$  the Research tuple is set to be active ( $b_i = true$ ).

**Definition 4.2.5** (Fast Forward Unit)

Let  $s_0 := (f_0, U_0, R_0, m_0, g_0)$  be a State and  $t_{ff} \in \mathbb{N}_0$  be a natural number including zero.

For all Unit tuples  $u = (a_i, b_i, (w_i, d_i), e_i, n_i) \in U_0$ , the corresponding Fast Forward Unit set  $U_{ff} \subset Units$  is defined as follows:

- If  $d_i > t_{ff}$ , then  $U_{ff} := \{(a_i, b_i, (w_i, d_i - t_{ff}), e_i, n_i)\}$
- If  $d_i < t_{ff}$ ,  $w \in ResearchTypes$ ,  $e > t_{ff}$ , then  $U_{ff} := \{(a_i, b_i, (IDLE, \infty), e_i - t_{ff}, n_i)\}$
- If  $d_i < t_{ff}$ ,  $w \in ResearchTypes$ ,  $e \leq t_{ff}$ , then  $U_{ff} := \{(a_i, b_i, (GasGathering, \infty), \infty, n_i)\}$
- If  $d_i < t_{ff}$ ,  $w \in UnitTypes$ ,  $e > t_{ff}$ ,  $partner(a_i, w_i) = true$ , then  $U_{ff} := \{(a_i, b_i, (IDLE, \infty), e_i - t_{ff}, n_i), (w_i, NOPARTNER, (IDLE, \infty), \infty, n_i)\}$
- If  $d_i < t_{ff}$ ,  $w \in UnitTypes$ ,  $e \leq t_{ff}$ ,  $partner(a_i, w_i) = false$ , then  $U_{ff} := \{(a_i, b_i, (GasGathering, \infty), \infty, n_i), (w_i, NOPARTNER, (IDLE, \infty), \infty, n_i)\}$
- If  $d_i < t_{ff}$ ,  $w \in UnitTypes$ ,  $e > t_{ff}$ ,  $partner(a_i, w_i) = true$ , then  $U_{ff} := \{(a_i, w_i, (IDLE, \infty), e_i - t_{ff}, n_i), (w_i, a_i, (IDLE, \infty), \infty, n_i)\}$
- If  $d_i < t_{ff}$ ,  $w \in UnitTypes$ ,  $e \leq t_{ff}$ ,  $partner(a_i, w_i) = false$ , then  $U_{ff} := \{(a_i, w_i, (GasGathering, \infty), \infty, n_i), (w_i, a_i, (IDLE, \infty), \infty, n_i)\}$

The Fast Forward Units mechanism  $FF^{units}$  calculates the changes on the set of Unit tuples during the next  $t_{ff}$  frames from the current state ( $s_0$ ):  $FF^{allunitsets}(s_0, t_{ff}) := \bigcup_{u \in U_0} U_{ff}$

The final set  $FF^{units}$  is then defined as the set that contains all units with the corresponding counters summed up, i.e.

$$FF^{units} := \{(a, b, t, e, n) \mid n = \sum_{i=1}^{|FF^{allunitsets}|} n_i, (a, b, t, e, n_i) \in FF^{allunitsets}\}$$

Therefore  $|FF^{units}| \geq |U_0|$ .

The set of Unit tuples after applying Fast Forward Unit mechanism is equal to the set before ( $U_0$ ) except, that all task and reservations have progressed by  $t_{ff}$ . This progression includes units becoming *IDLE* when they finished their task and starting gas gathering when their reservation is up. Also new Unit instances are added for finished Tasks with  $w \in UnitTypes$ . If the creating UnitType is in a PartnerRelation with the newly created, they become partners.

Having define all our helper functions we can now define the complete Fast Forward mechsanim.

**Definition 4.2.6** (Fast Forward Function)

Let  $s_0 := (f_0, U_0, R_0, m_0, g_0)$  be a State and  $t_{ff} \in \mathbb{N}_0$  be a natural number including zero. We define a function  $ff : States \times \mathbb{N}_0 \rightarrow States$  as  $ff(s, t_{ff}) = s'$  where  $s' = (FF^{time}(s_0, t_{ff}), FF^{units}(s_0, t_{ff}), FF^{research}(s_0, t_{ff}), FF^{mineral}(s_0, t_{ff}), FF^{gas}(s_0, t_{ff}))$

The function  $ff$  takes a *State*  $s$  and produces a *State*  $s'$  which is the result of the time progression by  $t_{ff}$ .

Churchill and Buro used another name for this function. They called it  $S' := Sim(S, \delta)$  where  $S, S'$  are states and  $\delta \in \mathbb{N}_0$  is the time.

### 4.3 Transitions

We will use *Quantified Requirement* to denote how many *Unit* tuples of the specified form  $(a, b)$  we describe.

**Definition 4.3.1** (Quantified Requirement)

We define a *Quantified Requirement* as the 3-tuple  $q = (a, b, n)$  where  $a, b \in UnitTypes$  and  $n \in \mathbb{N}$ .

The next definition corresponds to the action declaration Churchill and Buro defined. This however is not an action or operator in our transition system. Therefore we call it *Order*.

**Definition 4.3.2** (Order)

We define  $L_{Tech} = (Q_{Tech}, R_{Tech})$ ,  $L_{Consume} = (m_{Consume}, g_{Consume}, Q_{Consume})$  and  $v_{Producing} \in UnitTypes \cup ReserachTypes$ , where  $Q_{Borrow}, Q_{Tech}, Q_{Consume}, Q_{Reservation}, Q_{Producing}$  are finite sets of *Quantified Requirement* and  $R_{Tech}$  is a finite set of *ResearchTypes*.

With those ingredients we define an *Order*  $o$  as a 6-tuple:

$$o = (L_{Tech}, Q_{Borrow}, L_{Consume}, Duration, Q_{Reservation}, v_{Producing})$$

where  $Duration \in \mathbb{N}_0$ ,  $m, g \in \mathbb{R}$  and  $m, g \geq 0$ .

The set containing all defined *Orders* is denoted as *Orders*.

The informations necessary to define an *Order* tuple can be found in the BWAPI. All the information on *ResearchTypes* and *UnitTypes* that is needed to create all *Order* tuples for *StarCraft Brood War* can be found in the *Namespace References* of *BWAPI::UnitTypes* and *BWAPI::UpgradeTypes* ([https://bwapi.github.io/namespace\\_b\\_w\\_a\\_p\\_i\\_1\\_1\\_unit\\_types.html](https://bwapi.github.io/namespace_b_w_a_p_i_1_1_unit_types.html) respectively [https://bwapi.github.io/namespace\\_b\\_w\\_a\\_p\\_i\\_1\\_1\\_upgrade\\_types.html](https://bwapi.github.io/namespace_b_w_a_p_i_1_1_upgrade_types.html) is the direct link to the *Namespace References*).

We define the following orders for our definition:  $o_{Terran\_Command\_Center}$ ,  $o_{Terran\_Refinery}$ ,  $o_{Terran\_Supply\_Depot}$ ,  $o_{Terran\_Engineering\_Bay}$ ,  $o_{Terran\_SCV}$ ,  $o_{Terran\_Comsat\_Station}$ ,  $o_{Terran\_Infantry\_Weapons\_11}$ ,  $o_{Terran\_Infantry\_Armor\_11}$ . We explain the definition of  $o_{Terran\_Command\_Center}$  in detail, the other orders are defined analogues in the appendix A.

$$o_{Terran\_Command\_Center} = ((\{(Terran\_SCV, NOPARTNER, 1)\}, \emptyset), \\ \{(Terran\_SCV, NOPARTNER, 1)\}, (400, 0, \emptyset), 1896, \emptyset, \\ Terran\_Command\_Center)$$

The first entry represents, that the order  $o_{Terran\_Command\_Center}$  can only be issued when at least one unit of type *Terran\\_SCV* exists and that *Research* is

required to be active. The second entry represents how many *Unit* instances of which *UnitType* will be borrowed to perform the *Order*. In this case that would also be one SCV. The third entry is a tuple representing the mineral cost, gas cost and unit instances which will be consumed or “paid” when issuing the *Order*. The fourth entry represents the how many frames the order will need to finish. The fifth entry represents which and how many *Unit* instances will be marked as reserved when issuing the *Order*. The fifth entry represents the *UnitType* (or *ResearchType*) that will be build (or researched).

We can denote the set of all *Order* tuples *Orders* as follows:  $Orders := \{O_{Terran\_Command\_Center}, O_{Terran\_Refinery}, O_{Terran\_Supply\_Depot}, O_{Terran\_Engineering\_Bay}, O_{Terran\_SCV}, O_{Terran\_Comsat\_Station}, O_{Terran\_Infantry\_Weapons\_11}, O_{Terran\_Infantry\_Armor\_11}\}$ .

For the following definitions let  $s = (f, U, R, m, g)$  be a state and  $o = (L_{Tech}, Q_{Borrow}, L_{Consume}, Duration, Q_{Reservation}, v_{Producing})$  an order.

**Definition 4.3.3** (Tech Precondition)

Let  $s_0 := (f_0, U_0, R_0, m_0, g_0)$  be a State. Let  $o = (L_{Tech}, Q_{Borrow}, L_{Consume}, Duration, Q_{Reservation}, v_{Producing})$  be an order. Then we have  $L_{Tech} = (Q_{Tech}, R_{Tech})$  with the finite set of quantified requirements  $Q_{Tech} = \{q_1, \dots, q_k\}$  where  $k = |Q_{Tech}|$  and the finite set of researches  $R_{Tech}$ .

We define a function  $pre_{Tech} : States \times Orders \rightarrow \{true, false\}$  as  $pre_{Tech}(s, o)$ .

$pre_{Tech}(s, o)$  evaluates to true iff  $n_{Tech} \leq \sum_{\substack{(a,b,(w,d),e,n) \in U_0 \\ a=a_{Tech}, b=b_{Tech}}} n$  for all  $q_i = (a_{Tech}, b_{Tech}, n_{Tech})$  and  $b = true$  for all  $(v, b) \in R_0$  where  $v \in R_{Tech}$ .

The functions  $pre_{Borrow} : States \times Orders \rightarrow \{true, false\}$ ,  $pre_{Supply} : States \times Orders \rightarrow \{true, false\}$ ,  $pre_{Consume} : States \times Orders \rightarrow \{true, false\}$  and  $pre_{Reservation} : States \times Orders \rightarrow \{true, false\}$  are defined analogue to  $pre_{Tech}$ .

**Definition 4.3.4** (Precondition)

We define a function  $pre : States \times Orders \rightarrow \{true, false\}$  as  $pre(s, o)$ .  $pre(s, o)$  evaluates to true iff.  $pre_{Tech}(s, o)$  evaluates to true and  $pre_{Borrow}(s, o)$  evaluates to true and  $pre_{Supply}(s, o)$  evaluates to true and  $pre_{Consume}(s, o)$  evaluates to true and  $pre_{Reservation}(s, o)$  evaluates to true.

An *Order* can only be issued when all preconditions are met.

**Definition 4.3.5** (Applicability)

We define applicability as follows:  $o$  is applicable in  $s$  iff  $\exists c \in \mathbb{N}_0$  s.t.  $pre(ff(s, c), o)$  evaluates to true

We use applicability of Orders to represent whether the order can be issued form the current state, without issuing any other Order.

**Definition 4.3.6** (Earliest Time Function: When)

Let  $o$  be applicable in  $s$ . Let  $P$  be a set of time points where the precondition is satisfied:  $P = \{t \mid pre(ff(s, t), o) \text{ evaluates to true}\}$ .

We define a function  $When : States \times Orders \rightarrow \mathbb{N}_0$  as

$$\text{When}(s, o) := t_i \mid t_i \in P, (t_i \leq t_j) \forall t_j \in P$$

When is not defined on  $(s, o)$  where  $o$  is not applicable in  $s$ .

Whenever there is one time point, there are infinity many s.t.  $o$  is applicable in  $s$ . Churchill and Buroproposed the idea of only ever considering the earliest time point, as waiting cannot reduce the overall make span.

**Definition 4.3.7** (Consume)

We remind that  $L_{\text{Consume}} = (m_{\text{Consume}}, g_{\text{Consume}}, Q_{\text{Consume}})$ .

We define the intermediate results  $m', g' \in \mathbb{R}$  and  $U' \in \text{Units}$  as follows:

$$m' := m - m_{\text{Consume}}$$

$$g' := g - g_{\text{Consume}}$$

$U' := U \setminus \{(a, b, t, e, n)\} \cup \{(a, b, t, e, n - n_{\text{Consume}})\}$  for all  $(a, b, t, e, n) \in U$  where  $(a, b, n_{\text{Consume}}) \in Q_{\text{Consume}}$

We define a function  $\text{Consume} : \text{States} \times \text{Orders} \rightarrow \text{State}$  as

$\text{Consume}(\text{State}, \text{Order}) := (f, U', R, m', g')$  for an Order  $o$  which is applicable in the State  $s$ .

The function  $\text{Consume}$  reduces resources and unit instances in a state by the amount given by the order  $o$ . Intuitively, this is paying the price for the order.

**Definition 4.3.8** (Borrow)

We remind that  $L_{\text{Producing}} = (q_{\text{Producing}}, r_{\text{Producing}})$  with  $q_{\text{Producing}} = (a_{\text{Producing}}, b_{\text{Producing}}, n_{\text{Producing}})$ .

We define the intermediate result  $U' \in \text{Units}$  as follows:

$U' := U \setminus \{(a, b, t_0, e, n_0), (a, b, t_1, e, n_1)\} \cup \{(a, b, t_0, e, n_0 - n_{\text{Borrow}}), (a, b, t_1, e, n_1 + n_{\text{Borrow}})\}$  for all  $(a, b, t, e, n) \in U$  where  $(a, b, n_{\text{Borrow}}) \in Q_{\text{Borrow}}$ ,  $t_0 = (\text{IDLE}, \infty)$ ,  $t_1 = (a_{\text{Producing}}, \text{Duration})$  and  $e_0 > \text{Duration}$

We define a function  $\text{Borrow} : (\text{State}, \text{Order}) \rightarrow \text{State}$  as

$\text{Borrow}(\text{State}, \text{Order}) := (f, U', R, m, g)$  for an Order  $o$  which is applicable in the State  $s$ .

The  $\text{Borrow}$  function calculates the set of units where the order has been issued. Note that the number of unit instances will not be changed by this function.

**Definition 4.3.9** (Reservation)

We define the intermediate result  $U' \in \text{Units}$  as follows:

$U' := U \setminus \{(a, b, t, e_0, n_0), (a, b, t, e_1, n_1)\} \cup \{(a, b, t, e_0, n_0 - n_{\text{Reservation}}), (a, b, t, e_1, n_1 + n_{\text{Reservation}})\}$  for all  $(a, b, t, e, n) \in U$  where  $(a, b, n_{\text{Reservation}}) \in Q_{\text{Reservation}}$ ,  $e_0 = \infty$ ,  $e_1 = \text{Duration}$  and  $t = (w, d)$  with  $d < \text{Duration}$ .

We define a function  $\text{Reservation} : (\text{State}, \text{Order}) \rightarrow \text{State}$  as

$\text{Reservation}(\text{State}, \text{Order}) := (f, U', R, m, g)$  for an Order  $o$  which is applicable in the State  $s$ .



We use the *Reservation* function, to mark units. These marked units will be assign to GasGathering when the Refinery, which starts construction with the same order, is finished.

**Definition 4.3.10** (Action)

An Action is characterized by an Order  $o$  and an number  $t \in \mathbb{N}_0$ . An Action  $a$  is a 2-tuple  $a := (o, t)$

The set of all Actions is denoted as  $Actions := Orders \times \mathbb{N}_0$ .

For every Operator there exist infinity many Actions. When talking about the Build Order we are not really interested in which Action has been chosen. We want to know the Order that characterized each action. Therefore, when talking about Build Orders, paths or Actions, we can use the *UnitType* the Order of the Actions produces instead of the Action.

**Definition 4.3.11** (Transition)

Let  $s, s'$  each be a State and  $a = (o, t)$  an Action.

A transition  $s \xrightarrow{a} s'$  exists iff.  $o$  is applicable in  $s$ .

If  $o$  is applicable in  $s$ , then  $s' := Reservation(Borrow(Consume(ff(s, t), o), o), o)$

A transition can but doesn't have to change the framecounter (if resources, tech, etc are currently available, up to unlimited actions can be performed within the same frame - although in reality only every a few). If it does, all Tasks are advanced in time accordingly. Part of each transition is, to calculate the completion of Tasks and the resulting changes of the state.

## 4.4 Entry and Solution

The final part of our model is the definition of cost, initial states and goals.

**Definition 4.4.1** (Cost)

The cost of an Action  $a = (o, t)$  is defined as  $cost(a) := t$ .

We are searching for the a solution which minimizes the make span. The cost of an action is then the time by which the action advanced the time.

The search for Build Orders is usually started in the state the game is currently in, whichever that might be. As a game of *StarCraft Brood War* usually starts in the same state ( $i_0$ ), we will define it as the default initial state.

**Definition 4.4.2** (Initial State)

Any state  $s$  can be chosen as initial state.

We define the default initial state

$$i_0 = (0, \{(Terran\_SCV, \emptyset, (IDLE, \infty), 4), \\ (Terran\_Command\_Center, \emptyset, (IDLE, \infty), 1)\}, \{ \}, 50.0, 0.0)$$

We have reached a goal state, if in the *State* at least the *Researches* contained in the goal are active and the *State* contains at least the number of *Unit* instances the goal requires as well as at least the amount of resources in the goal.

**Definition 4.4.3** (Goal)

We define a Goal  $G$  as the 4-tuple  $G := (Q_G, R_G, m_G, g_G)$  where  $Q_G$  is a set of Quantified Requirement tuples,  $R_G$  a set of ReserachType labels and  $m_G, g_G \in \mathbb{R}$ .

We define the set of all Goal tuples as *Goals*.

Furthermore we define a function  $fulfil : States \times Goals \rightarrow \{true, false\}$  as

$$fulfil(s, G) := ((n_G \leq \sum_{u:=(a,b,(w,d),e,n),u \in U} n \mid a = a_G, b = b_G) \forall q \in Q_G) \wedge ((b = true \mid (v, b) \in R, v = r) \forall r \in R_G) \wedge (m \geq m_G) \wedge (g \geq g_G)$$

A state  $s$  can directly reach a Goal  $G$  iff there exists a  $c \in \mathbb{N}_0$  where  $fulfil(ff(s, c), G)$  evaluates to true.

A state  $s$  is a goal state iff  $fulfil(s, G)$  evaluates to true.

The make span of the resulting Build Order is then the frame count of the found goal state.

As example consider the *Goal*:

$$G_0 = (\{(Terran\_SCV, \emptyset, 4), (Terran\_Engineering\_Bay, \emptyset, 1)\}, \{Terran\_Infantry\_Weapons\_1\}, 0.0, 0.0)$$

The Goal  $G_0$  requires four SCV's, one Engineering Bay to exist and the *Research* with *ResearchType Terran\_Infantry\_Weapons\_1* to be active. As both *minerals* and *gas* cannot be negative, the requirements on them are trivially fulfilled for the Goal  $G_0$ .

**Definition 4.4.4** (Finishing Step)

Let  $G$  be a goal. Let  $s$  be a state who can directly reach  $G$ .

Let  $P$  be a set of time points where the goal is fulfilled:  $P = \{t \mid fulfil(ff(s, t), o)$  evaluates to true}

We define an action  $a_{FinishingStep}$  with the order  $o_{FinishingStep} := (G, \emptyset, (0, 0, \emptyset), 0, \emptyset, (\emptyset, \emptyset))$  and the time  $t_{FinishingStep} := t_i \in P$  where  $t_i \leq t_j$  for all  $t_j \in P$ .

The action *Finishing Step* is applicable iff.  $s$  can directly reach the goal  $G$ . The functions *Consume*, *Borrow* and *Reservation* will return the same state. Therefore applying *Finishing Step* can also be described as  $s' = ff(s, When(s, a))$ . Every Build Order that reaches the goal must include this action. However it just represents waiting until the orders are finished. Therefore when writing down a Build Order we will not include it.

The representation described by Churchill and Burois slightly different but of similar power. Instead of characterizing units as actors which have tasks, he

declares them as resources, an approach which was motivated by Chan et al. [1]. While their approach is further away from the actual game, it allows easier modelling in PDDL. Either representation can be linearly transformed into the other and is equally powerful.

The model shown contains a representative subset of the model for the full *Terran* race. For the other two races (*Zerg* and *Protoss*), the model is analogous.

## 4.5 Example

This is the state a game of *StarCraft Brood War* normally starts:

$$s_0 = (0, \{(Terran\_SCV, \emptyset, (IDLE, \infty), \infty, 4), \\ (Terran\_Command\_Center, \emptyset, (IDLE, \infty), \infty, 1)\}, \{\}, 50.0, 0.0)$$

One possible action is starting the construction of a *Engineering Bay*. For that we first need to collect some minerals, therefore the Fast Forward mechanism advances the time by 417 frames as part of applying the action.

$$s_1 = (417, \{(Terran\_SCV, \emptyset, (IDLE, \infty), \infty, 3), \\ (Terran\_SCV, \emptyset, (Terran\_Engineering\_Bay, 900), \infty, 1), \\ (Terran\_Command\_Center, \emptyset, (IDLE, \infty), \infty, 1)\}, \{\}, 0.06, 0.0)$$

Now we start the construction of a *Refinery* because we will need it for collecting *vespine gas* for the research later. Of course the algorithm does not specify which action to take. It could therefore likely be exploring other parts of the search tree first. We will now only look at a path leading to a goal directly.

$$s_2 = (1158, \{(Terran\_SCV, \emptyset, (IDLE, \infty), 600, 2), \\ (Terran\_SCV, \emptyset, (Terran\_Engineering\_Bay, 159), \infty, 1), \\ (Terran\_SCV, \emptyset, (Terran\_Refinery, 600), 600, 1), \\ (Terran\_Command\_Center, \emptyset, (IDLE, \infty), \infty, 1)\}, \{\}, 0.096, 0.0)$$

Here happened several things. In the last state there were two constructions going on. After applying the action  $a_{Terran\_Infantry\_Weapons\_1}$ , both constructions have finished. We now have two new *Units* (*Terran\_Engineering\_Bay*, *Terran\_Refinery*) in our state. Also 3 *SCVs* started gathering *Vespin Gas*. However the limiting precondition here was actually the minerals, because after it started, the gas gathering was much faster.

$$\begin{aligned}
s_3 = & (2338, \{(Terran\_SCV, \emptyset, (IDLE, \infty), \infty, 1), \\
& (Terran\_SCV, \emptyset, (GasGathering, \infty), \infty, 3), \\
& (Terran\_Engineering\_Bay, \emptyset, (Terran\_Infantry\_Weapons\_1, 4000), \infty, 1), \\
& (Terran\_Refinery, \emptyset, (IDLE, \infty), \infty, 1), \\
& (Terran\_Command\_Center, \emptyset, (IDLE, \infty), \infty, 1)\} \\
& , \{\}, 0.096, 0.0)
\end{aligned}$$

Now all we have to do is advance the time until the research is done. After applying the *Finish Step* action we are in the state  $s_4$ , which is a goal state.

$$\begin{aligned}
s_4 = & (6338, \{(Terran\_SCV, \emptyset, (IDLE, \infty), \infty, 1), \\
& (Terran\_SCV, \emptyset, (GasGathering, \infty), \infty, 3), \\
& (Terran\_Engineering\_Bay, \emptyset, (IDLE, \infty), \infty, 1), \\
& (Terran\_Refinery, \emptyset, (IDLE, \infty), \infty, 1), \\
& (Terran\_Command\_Center, \emptyset, (IDLE, \infty), \infty, 1)\} \\
& \{Terran\_Infantry\_Weapons\_1\}, \\
& 180.041, 840.8)
\end{aligned}$$

The Build Order for this example is then:

*Terran\_Engineering\_Bay, Terran\_Refinery, Terran\_Infantry\_Weapons\_1*

The make span of this Build Order is 6338 frames.

## 5 Experiments

While building our formalism we also implemented stand alone tool featuring a basic version of the algorithm 1 proposed by Churchill and Buro. Our implementation is capable of producing correct Build Order. However the implementation is not real-time. Churchill and Buro describe two admissible heuristics which the algorithm uses to bound the search depth. Our implementation misses those heuristics. As a result, we are much slower and cannot compute deep Build Orders. Also we have to use a manually computed initial bound. We removed the time limitation for our experiments to give perspective of the current power of the implementation.

We build our own test set, as we were not able to find an existing one. The table 1 shows results for Build Order goals from the default initial state  $i_0$ . We used short “SD” for *Supply Depot* and “EB” for *Engineering Bay*. The goals

goal	bound	makespan	Build Order	expanded	generated	time(ms)
7 SCV	1000	900	SCV,SCV,SCV	361	3152	979
7 SCV	1500	900	SCV,SCV,SCV	361	3152	978
1 SD	1500	974	SD	2107	16878	6199
1 EB	2000	1413	EB	29701	238018	107108
1 SD, 7 SCV	2000	1587	SCV,SCV,SCV,SD	10603	87933	36014

Table 1: testresults

are relatively simple as they only require a few more units, which all only cost *minerals* and have no other precondition than one *SCV* (which is given in  $i_0$  already).

Consider for example the goal “7 SCV”. Formally this is the goal  $G_0 = (\{(Terran\_SCV, \emptyset, 7)\}, \{\}, 0.0, 0.0)$ . To reach this goal we only need to build 7 *SCV*’s with a total *minerals* cost of 150.

The bound is the search depth in frames, any state with a make span greater than the bound will not be reached. The make span is also given in frames. The Build Order from left to right give the action sequence of the solution found. We used short “expanded” for the number of expanded nodes and “generated” equivalently for generated nodes. The solutions can easily be checked with our model and are correct.

## 6 Conclusion

In this thesis, we have introduced a formal model for Build Order search in *StarCraft Brood War* based on the intuition proposed by Churchill and Buro [2]. Our model features numerical values and handles temporal actions. We also implemented a stand alone planer as a prove of concept. Our planner is not very strong as it lacks algorithmic optimizations, but is capable of finding correct solutions for Build Order for the *Terran* race.

Our abstraction is very specialized on the game *StarCraft Brood War*. We belief it can be used for the successor *StarCraft 2* with only minor adaptation, because the two games are very similar.

In the future, we would like to adapt the model to more general RTS games. The abstraction used by Churchill and Buro [2] is a reasonable trade of between speed and optimality in *StarCraft Brood War*. However it can not handle other RTS games like *Age of Empires 2* where there are 4 different resources and the collection rate can vary. But the action applicability abstraction proposed, might also be used for resource collection. Instead of using single collection-actions which would dominate the search space or just fixed workers per resource policies which can be extremely suboptimal, we could define collecting-goals. These could just be the resources needed for a next unit. With the domain knowledge we have, we can derive the time needed for all preconditions besides the resources required. Then we can determine, how fast we need to collect the resources so that we have them available when the other preconditions are met, or as soon afterwards as possible.

## References

- [1] Hei Chan, Alan Fern, Soumya Ray, Nick Wilson, and Chris Ventura. Online planning for resource production in real-time strategy games. In *ICAPS*, pages 65–72, 2007.
- [2] David Churchill and Michael Buro. Build order optimization in starcraft. In *AIIDE*, pages 14–19, 2011.
- [3] Patrick Eyerich, Robert Mattmüller, and Gabriele Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *ICAPS*, pages 130–137, 2009.
- [4] Maria Fox and Derek Long. Pddl2. 1: An extension to pddl for expressing temporal planning domains. *JAIR*, 20:61–124, 2003.
- [5] Malte Helmert. 5. state-space search: State spaces. In *Foundation of Artificial Intelligence*, pages 9–16, 2016.
- [6] Opensource. BWAPI an api for interacting with starcraft: Broodwar (1.16.1). <http://bwapi.github.io/>.
- [7] Vinyals Oriol. Deepmind and blizzard to release starcraft 2 as an ai research environment. <https://deepmind.com/blog/deepmind-and-blizzard-release-starcraft-ii-ai-research-environment/>, 2016.

## A Complete List of Orders in our Model

$$\begin{aligned} o_{Terran\_Command\_Center} = (&(\{(Terran\_SCV, NOPARTNER, 1)\}, \emptyset), \\ &\{(Terran\_SCV, NOPARTNER, 1)\}, (400, 0, \emptyset), 1896, \emptyset, \\ &Terran\_Command\_Center) \end{aligned}$$

$$\begin{aligned} o_{Terran\_Refinery} = (&(\{(Terran\_SCV, NOPARTNER, 1)\}, \emptyset), \\ &\{(Terran\_SCV, NOPARTNER, 1)\}, (100, 0, \emptyset), 696, \\ &\{(Terran\_SCV, NOPARTNER, 3)\}, Terran\_Refinery) \end{aligned}$$

$$\begin{aligned} o_{Terran\_Supply\_Depot} = (&(\{(Terran\_SCV, NOPARTNER, 1)\}, \emptyset), \\ &\{(Terran\_SCV, NOPARTNER, 1)\}, (100, 0, \emptyset), 696, \emptyset, \\ &Terran\_Supply\_Depot) \end{aligned}$$

$$\begin{aligned}
o_{\text{Terran\_Engineering\_Bay}} = & ((\{( \text{Terran\_SCV}, \text{NOPARTNER}, 1), \\
& (\text{Terran\_Command\_Center}, \text{NOPARTNER}, 1)\}, \emptyset), \\
& \{( \text{Terran\_SCV}, \text{NOPARTNER}, 1)\}, (125, 0, \emptyset), 996, \emptyset, \text{Terran\_Engineering\_Bay})
\end{aligned}$$

$$\begin{aligned}
o_{\text{Terran\_SCV}} = & ((\{( \text{Terran\_Command\_Center}, \text{NOPARTNER}, 1)\}, \emptyset), \\
& \{( \text{Terran\_Command\_Center}, \text{NOPARTNER}, 1)\}, (50, 0, \emptyset), 300, \emptyset, \text{Terran\_SCV})
\end{aligned}$$

$$\begin{aligned}
o_{\text{Terran\_Comsat\_Station}} = & ((\{( \text{Terran\_Command\_Center}, \text{NOPARTNER}, 1)\}, \emptyset), \\
& \{( \text{Terran\_Comsat\_Station}, \text{NOPARTNER}, 1)\}, (50, 50, \emptyset), \\
& 600, \emptyset, \text{Terran\_Comsat\_Station})
\end{aligned}$$

Here the units required to be allowed to build a *Terran\_Comsat\_Station* has been reduced to the units contained in the model.

$$\begin{aligned}
o_{\text{Terran\_Infantry\_Weapons\_11}} = & ((\{( \text{Terran\_Engineering\_Bay}, \text{NOPARTNER}, 1)\}, \\
& \emptyset), \{( \text{Terran\_Engineering\_Bay}, \text{NOPARTNER}, 1)\}, (100, 100, \emptyset), \\
& 4000, \emptyset, \text{Terran\_Infantry\_Weapons\_11})
\end{aligned}$$

$$\begin{aligned}
o_{\text{Terran\_Infantry\_Armor\_11}} = & ((\{( \text{Terran\_Engineering\_Bay}, \text{NOPARTNER}, 1)\}, \\
& \emptyset), \{( \text{Terran\_Engineering\_Bay}, \text{NOPARTNER}, 1)\}, (100, 100, \emptyset), \\
& 4000, \emptyset, \text{Terran\_Infantry\_Armor\_11})
\end{aligned}$$

**Erklärung zur wissenschaftlichen Redlichkeit**

(beinhaltet Erklärung zu Plagiat und Betrug)

Bachelorarbeit / ~~Masterarbeit~~ (nicht Zutreffendes bitte streichen)

Titel der Arbeit (Druckschrift):

A Formalism for Build Order  
Search in StarCraft Brood War

Name, Vorname (Druckschrift): Wyss Severin

Matrikelnummer: 13-053-707


Hiermit erkläre ich, dass mir bei der Abfassung dieser Arbeit nur die darin angegebene Hilfe zuteil wurde und dass ich sie nur mit den in der Arbeit angegebenen Hilfsmitteln verfasst habe.

Ich habe sämtliche verwendeten Quellen erwähnt und gemäss anerkannten wissenschaftlichen Regeln zitiert.

Diese Erklärung wird ergänzt durch eine separat abgeschlossene Vereinbarung bezüglich der Veröffentlichung oder öffentlichen Zugänglichkeit dieser Arbeit.

ja  nein

Ort, Datum: Zwingen 1.12.2016

Unterschrift: 

*Dieses Blatt ist in die Bachelor-, resp. Masterarbeit einzufügen.*