# Predicting Opponent's Production in Real-Time Strategy Games with Answer Set Programming

Marius Stanescu and Michal Čertický

*Abstract*—**The adversarial character of real-time strategy (RTS) games is one of the main sources of uncertainty within this domain. Since players lack exact knowledge about their opponent's actions, they need a reasonable representation of alternative possibilities and their likelihood. In this article we propose a method of predicting the most probable combination of units produced by the opponent during a certain time period. We employ a logic programming paradigm called Answer Set Programming, since its semantics is well suited for reasoning with uncertainty and incomplete knowledge. In contrast with typical, purely probabilistic approaches, the presented method takes into account the background knowledge about the game and only considers the combinations that are consistent with the game mechanics and with the player's partial observations. Experiments, conducted during different phases of StarCraft: Brood War and Warcraft III: The Frozen Throne games, show that the prediction accuracy for time intervals of 1-3 minutes seems to be surprisingly high, making the method useful in practice. Root-mean-square error grows only slowly with increasing prediction intervals – almost in a linear fashion.**

*Index Terms*—**Real-Time Strategy, Answer Set Programming, prediction, StarCraft, WarCraft, opponent modelling, RTS.**

## I. Introduction

REAL-TIME STRATEGY (RTS) games represent a genre of video games in which players must manage economic tasks like gathering resources or building new bases, increase their military power by researching new technologies and training units, and lead them into battle against their opponent(s). RTS games serve as an interesting domain for Artificial Intelligence (AI) research, since they represent well-defined complex adversarial systems [3] which pose a number of interesting AI challenges in the areas of planning, learning, spatial/temporal reasoning, domain knowledge exploitation, task decomposition and – most relevant to the scope of this article – dealing with uncertainty [15].

One of the main sources of uncertainty in RTS games is their adversarial nature. Since players do not possess an exact knowledge about the actions that their opponent will execute, they need to build a reasonable representation of possible alternatives and their likelihood. This work specifically tackles the problem of predicting opponent's unit production. The method presented here allows artificial players (bots) playing RTS games to estimate the number of different types of units produced by their opponents over a specified time period.

Various problems related to uncertainty and opponent behaviour prediction in RTS games have already been addressed in recent years (an extensive overview can be found in [15]). For example, Weber and Mateas [21] proposed a data mining approach to strategy prediction, Dereszynski et al. [5] used Hidden Markov Models to learn the transition probabilities within building construction sequences, Synnaeve and Bessiére [17] presented a Bayesian semi-supervised model for predicting game openings (early game strategies). Weber et al. [22] used a particle model to track opponent's units out of vision range and Kabanza et al. [13] used hierarchical task networks to recognize opponent's intentions, while extending the probabilistic hostile agent task tracker (PHATT) by [10]. We supplement this work by introducing a logic-based solution to yet another subproblem from this area – predicting opponent's unit production. This particular challenge is relevant to any conventional RTS game and takes almost the same form in all games of this genre. Typically, the differences are only in the input data: costs and production times of individual unit types and their technological preconditions. Reasoning behind the prediction task itself is, however, not dependent on game-specific information.

Expert knowledge about complex domains, such as RTS games, is often extensive and hard-coding the reasoning for a number of different games in imperative programming languages tends to be quite inconvenient and time-consuming. Therefore it makes sense to consider using *declarative knowledge representation paradigms*, many of which are well-suited for this kind of problems.

Compared to traditional imperative methods, a declarative approach allows for easier expression of typical RTS mechanics. Game-specific knowledge can be separeted from common reasoning mechanisms by employing modular design patterns. A specific paradigm of logic programming called *Answer Set Programming* (ASP) [11], [12], [2] was selected for the solution presented in this work. Its nonmonotonic semantics with negation as failure is well suited for reasoning with uncertainty and incomplete knowledge, and has already been used for prediction tasks outside the computer games domain [6], [8]. In contrast with purely probabilistic approaches, using ASP to predict opponent's unit production allows for the exploitation of background knowledge about the game, so that only those unit combinations that are reliably consistent with game's rules and player's partial observations (both encoded by ASP) are considered.

## II. Answer Set Programming

Answer Set Programming has lately become a popular declarative problem solving paradigm with a growing number of applications. In the area of game AI, ASP has been used

M. Stanescu is with the GAMES Group at University of Alberta in Edmonton. e-mail: astanesc@ualberta.ca

M. Čertický is with the Agent Technology Center at Czech Technical University in Prague. e-mail: certicky@agents.fel.cvut.cz

in attempts to implement a "general player", either in combination with other techniques [14], or in ASP-only projects like the one presented in [1]. Additional work has been done on translating the general Game Description Language (GDL) into ASP [19]. However, to the best of our knowledge, the only published application of ASP specifically for RTS gameplay dealt with wall-in building placement in StarCraft [20].

The original language associated with ASP allows the formalization of various kinds of common sense knowledge and reasoning, including constraint satisfaction and optimization. The language is a product of a research aimed at defining a formal semantics for logic programs with default negation [11], and was extended to allow also a classical (or *explicit*) negation [12] in 1991. An ASP *logic program* is a set of *rules* of the following form:

$$h \leftarrow l_1, \dots, l_m,\ not\ l_{m+1}, \dots\ not\ l_n.$$

where $h$ and $l_1, \dots, l_n$ are classical first-order-logic literals and $not$ denotes a default negation. Informally, such a rule means that "if you believe $l_1, \dots, l_m$, and have no reason to believe any of $l_{m+1}, \dots, l_n$, then you must believe $h$". The part to the right of the "$\leftarrow$" symbol ($l_1, \dots, l_m,\ not\ l_{m+1}, \dots\ not\ l_n$) is called the *body*, while the part to the left of it ($h$) is called the *head* of the rule. Rules with an empty body are called *facts*. Rules with empty head are called *constraints*. It is said that a literal is *grounded* if it is variable-free. A logic program is called *grounded* if it contains only grounded literals.

The ASP semantics is built around the concept of *answer sets*. Consider a grounded logic program $\Pi$ and a consistent set of classical grounded literals $S$. A subprogram called *program reduct of* $\Pi$ *w.r.t. set S* (denoted $\Pi^S$) can then be obtained by removing each rule that contains $not\ l$, such that $l \in S$, in its body, and by removing every "$not\ l$" statement, such that $l \notin S$. A set of classical grounded literals $S$ is *closed* under a rule $a$, if holds $body(a) \subseteq S \Rightarrow head(a) \in S$.

*Definition 1 (Answer Set):* Let $\Pi$ be a grounded logic program. Any minimal set $S$ of grounded literals *closed* under all the rules of $\Pi^S$ is called an *answer set* of $\Pi$.

Intuitively, an answer set represents one possible meaning of knowledge encoded by a logic program $\Pi$ and a set of classical literals $S$. Typically, one answer set *corresponds to one valid solution of a problem* encoded by the logic program.

In this work, returned answer sets will correspond to different combinations of units that might be trained by the opponent during the following time period, which are consistent with the game rules and current partial observations.

## III. Predicting Unit Production

This work assumes that modeled game mechanics are consistent with the following conventions, typical for RTS games:

- Units are trained in production facilities (specific types of structures; e.g. Barracks, Workshop, Gateway).
- Producing units or structures requires resources and time.
- There is a continuous income of consumable resources (e.g. minerals, gold) that can be approximated based on a number of certain units (referred to as workers).
- Renewable resources (e.g. supply, accommodation or power) are obtained by constructing certain structures.
- Certain unit types may have specific technological prerequisites (availability of certain structures or technologies).

Good examples of games that satisfy all of these conditions can be found in the *Age of Empires*, *WarCraft* or *Command & Conquer* franchises. Implementations presented in this article use two such games: *StarCraft: Brood War* and *WarCraft III: The Frozen Throne*, both developed by Blizzard Entertainment.

Since the goal is to predict the most probable among the valid combinations of units, the following need to be accomplished:

- generating all the unit combinations that could be trained, given the opponent's observed production facilities,
- removing the unfeasible combinations (units take time to train, and they cost resources and supply),
- selecting the most probable valid combination, using the information from recorded games.

The game was accessed in real time via BWAPI[1] in case of StarCraft and by custom replay parser in case of WarCraft III. At each relevant time step, a logic program representing the partial knowledge about current game situation was constructed and the answer sets of this program were computed using the ASP solver called CLASP [9], which supports several convenience features like generator rules, aggregates or optimization statements. When reading the code fragments in this section, one should note that the parameters starting with uppercase letters are variables (e.g. "`T`", "`Z`", "`Number`"), while the others are constants corresponding to objects from the domain or numbers (e.g. "`gateway`" or "`40`"). The following examples focus on the StarCraft game, and therefore StarCraft-specific terminology is used, but only for the names of units and structures; the proposed method can be easily converted for other RTS games.

### A. Generating Unit Combinations

First of all, the maximum number of each type of unit that can be trained using the opponent's production facilities needs to be computed. For example, a Gateway can be used to train Zealots (each takes 40 seconds) or Dragoons (50 seconds). For the sake of simplicity, only these two types of units are considered in the examples below. Over a time interval of 200 seconds, the maximum number of Zealots that can be trained will be 5, and the maximum number of Dragoons 4. If the opponent had two Gateways instead, then these numbers would double, to 10 and 8 respectively. For example, the code below computes the maximum number of Zealots:

```
timeGateway(Number*T)  :- built(gateway,Number), interval(T).
maxZealots(Z)          :- timeGateway(T), Z = @min(T/40,24).
```

Computation time generally grows exponentially with the maximum number of allowed units, so certain limits were imposed (in the above case, *maxZealots* was limited to 24).

Note that this maximum number represents how many units can be trained independently of other units, so obviously it is

[1]`http://github.com/bwapi/bwapi`

not possible to have 4 Zealots and 3 Dragoons within 200 seconds, using a single Gateway. All unit combinations can be generated using the following code – however, many of them will not be feasible:

```
1 {zealots(0..N)}  1    :- maxZealots(N) , N > 0.
1 {dragoons(0..N)} 1    :- maxDragoons(N) , N > 0.
gateway_units(Z,D):- zealots(Z), dragoons(D).
```

### B. Removing Invalid Combinations

Next, the combinations that exceed the time limit need to be removed using the following constraint:

```
:- gateway_units(Z,D), timeGateway(T1), (Z*40 + D*50) >= T1.
```

Assuming that the opponent tries to spend everything as soon as possible, the maximum quantity of resources available over the next time period can be accurately approximated using the number of current workers. The *resources per second* gathering rate for a worker is a well known constant. We assume that the opponent continues producing workers at a steady rate, which is what skilled players usually do in StarCraft.

Knowing the maximum amount of resources, more constraints can be added to prune out all the combinations that are too expensive to train. However, there is one more restriction to be imposed, concerning the *units supply*. For example, all the Gateway units take up 2 supply each, and a player cannot exceed the current population limit. However, by building Pylons (specific structures that can be built for 100 minerals), a player raises the supply cap by 8 – and 4 extra Zealots can be trained, for example. If a combination uses more supply than is currently available to the player, building as many Pylons as necessary for sustaining it is also imposed.

### C. Choosing the Most Probable Combination

The previous subsections have shown how to produce many possible unit combinations – answer sets representing different valid armies which can be trained over the selected time interval. However, the goal is to be able to accurately *predict the most probable* such *combination*. Consequently, only one answer set should be selected from all the possibilities. One reasonable approach is to assume that the opponent tries to spend the maximum amount of resources or, equivalently, to have as few remaining resources as possible. This can be easily accomplished using CLASP's *#minimize* statement.

Alternatively, it might be useful to predict the most dangerous valid unit combinations, so the player can prepare against them in particular. However, this requires quantifying the strength of a group of units relative to other units or states which is a hard problem, out of scope of this article (simulations [4] or machine learning [16] might help).

Finally, the approach chosen in this paper is to actually pick the *most probable* unit combination for a given game phase, based on a set of recorded games (replays[2]). Thus, the goal is to output the army combination most likely to be seen from the opponent, or at least sort the answer sets according to such a criterion. The first step is to compute the probability of the opponent having *Z* zealots, *D* dragoons, and so on, at a specific *Time*, denoted $P(Z, D, \cdots, Time)$. Then, the answer sets are sorted according to this value. An obvious simplification considered was to assume that training a unit is *independent* of training the others. With this simplification, one can express the probability of the whole unit combination at a given time as:

$$P(Z, D, \cdots, Time) = P(Z, Time) \cdot P(D, Time) \cdot \ldots$$

Finding the probability of the opponent having *Z* zealots, etc. at a specific time is an easy task, accomplished by parsing a large set of replays and computing it as follows:

$$P(Z, T) = \frac{\text{Nr. of replays with exactly Z zealots trained at time T}}{\text{Total nr. of replays}}.$$

Even thought this simplifying assumption is quite strong, as there are considerable correlations between training certain unit types, it will be shown that it can be used for prediction with sufficient accuracy (see Section IV). Once able to compute the probability of specific unit combination, CLASP's optimization statements are used to enforce the selection policies discussed above:

```
    #maximize [p(X) = X @ 2].
    #minimize [remainingRes(M,G) = M+G @ 1 ].
```

The implementation used for the experiments combines two selection policies. First and foremost, it maximizes the likelihood of the combination (priority 2). If there are two equally probable answer sets, the one that leaves the opponent with less remaining resources (M – minerals and G – gas) is favoured (priority 1).

## IV. Experiments and Results

This section describes the conducted experiments and explains the output of the implemented program. Afterwards, the metrics used for quantifying the prediction accuracy are introduced, and the obtained results are presented and discussed.

To the best of our knowledge, there have been no other published solutions to the problem of opponent's unit production, and so it is impossible to empirically compare the performance of the implemented method to alternatives.

### A. Experiment Design

The experiments were conducted in four different game times: at $5^{th}$, $10^{th}$, $20^{th}$ and $30^{th}$ minute after the start of the game for StarCraft and at $5^{th}$, $10^{th}$, $15^{th}$ and $20^{th}$ minute for WarCraft III. The first time is usually associated with the early-game phase, next two with mid-game and the last one is usually classified as a late-game phase. For each experiment, the units the opponent will have 1, 2 and 3 minutes in the future are predicted. 300 StarCraft and 255 WarCraft III replays were used for the evaluation. However, some games took less than 30, 20, or even 10 minutes to finish. For example, in the last experiment ($30^{th}$ minute), only 65 out of the initial 300 StarCraft replays were still valid, and only

[2]Note that these replays should be selected carefully. For example, if the probabilities were computed using mostly professional human vs. human games, the prediction accuracy against beginners or bots might suffer. We worked with the replays of high-level human vs. human StarCraft games from [18] and a collection of replay packs from different WarCraft III tournaments.

34 WarCraft III replays were valid after 20 minutes (a replay is considered valid if at least one unit is trained after the end of the prediction time).

Below is a sample output of the program, representing a three-minute prediction from $10^{th}$ minute of a StarCraft game. It took less than 60ms to compute, and it returned four answer sets. Below the prediction, there are the actual units trained by the player in this interval – 7 Dragoons (second unit type trained in the Gateway). Note that it is needed to keep the optimization scores and probabilities unnormalized, because CLASP can only work with integers. For instance, answer set 1 below has $prob = 5312104$ which corresponds to a probability of $p = 0.00053$. The optimization score the program is trying to minimize is just a constant minus the unnormalized probability.

```
------- Predicting for player 0 time 9000 to 11700 --------
        ( 10 mins, after 3 )

---------------------- CLASP output -----------------------
clasp version 2.1.1 Solving...
Answer: 1
gateway_units(4,4,0,0) robotic_units(0,0,0)
stargate_units(0,0,0,0) remainingRes(2202,866) prob(5312104)
Optimization: 112226583
Answer: 2
gateway_units(1,4,0,0) robotic_units(0,0,0)
stargate_units(0,0,0,0) remainingRes(2602,866) prob(5720728)
Optimization: 111817959
Answer: 3
gateway_units(0,7,0,0) robotic_units(0,0,0)
stargate_units(0,0,0,0) remainingRes(2227,716) prob(6803028)
Optimization: 110735659
Answer: 4
gateway_units(1,6,0,0) robotic_units(0,0,0)
stargate_units(0,0,0,0) remainingRes(2252,766) prob(7266870)
Optimization: 110271817
OPTIMUM FOUND

Models      : 1
  Enumerated: 4
  Optimum   : yes
Optimization: 110271817
Time        : 0.053s
CPU Time    : 0.031s

---------------- Units trained in reality ----------------
 Gateway:  0 7 0 0
 Robotics: 0 0 0
 Stargate: 0 0 0 0
```

## B. Evaluation Metrics

The number of units a player trains between two time stamps is predicted. There are 11 different types of units in the Protoss faction (4 from the Gateway and Stargate each, and 3 from the Robotics Bay), so the obtained answer set is converted to a 11-dimensional vector. To keep things easy to understand, the examples below only show the first 4 elements of this vector – the units produced from the Gateway. Assume that the real number of trained units is shown on the first line, and that the prediction for this situation is the second line:

| Zealots | Dragoons | High Templars | Dark Templars | 7 more |
|---|---|---|---|---|
| 7 | 3 | 0 | 2 | ··· |
| 5 | 3 | 0 | 3 | ··· |

The difference between these two has to be computed in order to obtain an accuracy measure. The *root-mean-square error* (RMSE) was chosen, a very popular tool for this type of tasks. It is frequently used to measure differences between values predicted by a model or an estimator and the values actually observed:

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{n}(x_{1,i} - x_{2,i})^2}{n}}$$

where n is a number of values predicted, $x_{1,i}$ are the real values observed and $x_{2,i}$ are the predictions.

Applied to the previous example:

$$\text{RMSE}_{1:4} = \sqrt{\frac{(7-5)^2 + (3-3)^2 + (0-0)^2 + (2-3)^2}{4}} = 1.118$$

If the prediction is off by one unit for *every* unit type, then a resulting RMSE of 1 is obtained. But, as presented in the example, if the prediction is wrong by more units (two zealots), even if most of the others are exactly right, a RMSE bigger than 1 is obtained. This happens because RMSE penalizes wrong predictions more, the more wrong they are, by squaring the difference. With this in mind, a RMSE of approximately 1 for the accuracy on all eleven units is more than decent and can be caused for example:

- by being 1 unit wrong for all unit types,
- or at the extreme, being right for 10 unit types and wrong for the last one by 3.

In the example presented in the previous subsection, the output consisted of 4 answer sets in decreasing order w.r.t. the CLASP optimization score. Their RMSE values are 1.5075 (answer set 1), 0.9534 (answer set 2), 0 (answer set 3) and 0.4264 (answer set 4).

The last answer set printed (nr. 4) has the best optimization score, and is called the *optimal* answer set. Since lower optimization score does not necessarily mean better RMSE, the *best* answer set is also recorded – the one with the lowest RMSE (which in the example is answer set nr. 3). This is the closest one could get to the values actually observed, assuming a perfect job with the optimization statements and heuristics.

As a simple illustrative baseline to compare against, the trivial per-unit average (AVG) over all the valid replays was used, with some minor modifications. For example, for a target prediction the average across the replays is 5 Zealots and 2 Dragoons for a total of $7 \times 2 = 14$ supply. However, if in the current game the player does not have the required structure for producing Dragoons, the AVG prediction is modified to 7 Zealots and 0 Dragoons instead, to maintain the average target supply of 14. Naturally, the proposed method is expected to achieve higher accuracy than this baseline, since it takes into account background knowledge and partial observations.

## C. Results

The algorithm had been run for all 300 StarCraft and 250 WarCraft III replays, then the values for three quantities of interest were averaged: RMSE of AVG baseline, best and optimal answer sets. The results are shown in figure 1 for StarCraft and figure 2 for WarCraft III . Standard deviation divided by the square root of the number of examples is shown as the shaded quantity in the plots. This measure is known as *standard error*; one margin of standard error gives 68% confidence (of the result being within the shaded area).

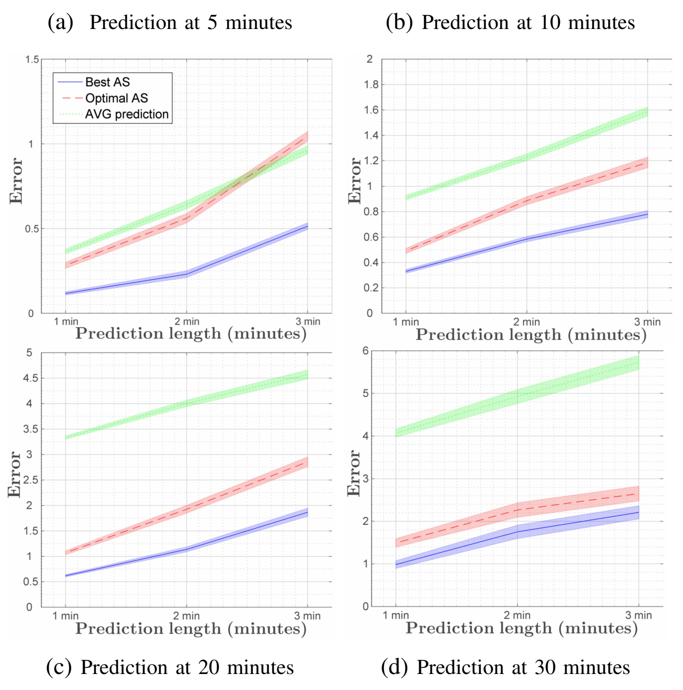


Fig. 1: Average predictions over StarCraft replays, shown as RMSE for best and optimal answer set, as well as a baseline that predicts the average number of units trained across all replays. The number of replays varies from 300 ($5^{th}$ min) to 65 ($30^{th}$ min).

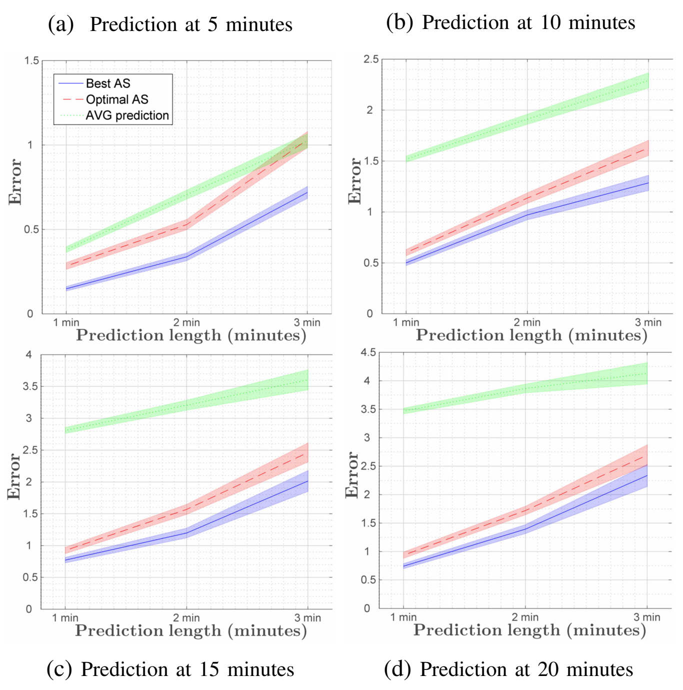


Fig. 2: Average predictions over WarCraft 3 replays, showing the same quantities as Figure 1. The number of replays varies from 255 ($5^{th}$ min) to 34 ($20^{th}$ min).

As expected, the best answer set has a significantly lower error than the rest. However, the longer the game, the closer the optimal answer set gets to the best answer set. Also, the deviation is larger as the time increases, partly because there are more answer sets computed, and partly due to the fact that the replay training set is smaller.

Both optimal and best answer sets have much lower RMSE than the AVG baseline, except for the *early-game* prediction at $5^{th}$ minute. At this time, there are many instances in which no fighting units are trained, or they are produced just seconds after the prediction range, which makes the prediction more difficult. While it is likely that the correct answer set is among the ones printed by CLASP (best answer set still has significantly lower RMSE), the optimal answer set chosen is quite close to the baseline in this case. Biasing the algorithm to choose the unit combinations that maximize the amount of resources spent seems to be counter-productive in this phase.

The most important area is around $10^{th}$ to $20^{th}$ minute – the *mid-game*. Before that, the game is dominated by the build order prediction and is more strategy oriented. After 20 minutes, many matches are finished already or the winner is obvious, and players do not train units so much. The prediction error at $10^{th}$ minute (until $11^{th}$, $12^{th}$ and $13^{th}$ min) is around 0.5, 0.9 and 1.2 for StarCraft and 0.6, 1.1 and 1.6 for WarCraft III, which is a satisfactory result. Predictions at this time tend to be more difficult in WarCraft III, where player's resource spending patterns temporarily deviate from typical RTS: they hire their second "hero" and purchase an equipment for him/her. However, the errors for the last two game times are very similar for StarCraft and WarCraft III, the RMSE being close to 2 for the 2 minute prediction and a little under 3 for the 3 minute predictions. This level of accuracy could certainly be useful in a real game and help the AI make better decisions towards what units to train to beat the adversary.

## D. Running Times

For StarCraft, the running times of the prediction are on average around 10ms, 100ms, 2s and 8s for the four game phases ($5^{th}$, $10^{th}$, $20^{th}$ and $30^{th}$ minute). This happens because after 30 minutes a player may have more than 10 production facilities, resulting in many valid combinations. Usually 20% of the time is spent either grounding or reading the grounded program, especially as the prediction time increases. A 10 second cut-off was imposed for the CLASP solver; we force the program to terminate after 10 seconds and return the best among the answer sets it managed to find so far.

The same prediction problem could in theory be solved using a "brute-force" approach with identical results in some kind of imperative programming language by (1) looping over all the unit combinations, (2) checking the validity of each combination, (3) assigning the probability to it and (4) selecting the most probable one. This in fact resembles the procedure performed by the CLASP ASP solver: (1) looping over the answer set candidates invoked by the generator rules, (2) eliminating the candidates that violate some of the constraints, (3) computing the optimization score and (4) selecting the best answer set according to it. Even though both of these problems have a high worst-case time complexity, ASP solvers like CLASP are quite fast in the average case, since they employ a variety of heuristic search strategies in the process, including *BerkMin*-like, *Siege*-like or *Chaff*-like decision heuristics [9] and can reduce the search space using the optimization statements.

## V. Conclusion

A prediction system based on the Answer Set Programming paradigm was successfully built and its performance evaluated. As experiments suggest, it can accurately predict the number and type of units a StarCraft or WarCraft III player trains in a given amount of time. Very good results were demonstrated during mid-game and late-game ($10^{th}$ to $20^{th}$ minute), for prediction length of 1 to 3 minutes. This is the interval in which the prediction of opponent's army production would be most useful.

The presented implementation is faster than expected from an ASP program. For the targeted interval described above, the running times averaged around 200ms, which is acceptable for a high-level decision making in RTS games AI.

It is clear from described results that building a simple probabilistic framework works with decent results, and that using ASP in strategy games brings certain advantages. A few other interesting tasks, suitable for future work, would be:

- Smoothing over the probabilities or working with a Gaussian framework. We currently only consider $P(Zealots, Time)$, but looking from $Time - \Delta t$ to $Time + \Delta t$ and using weights, or trying to fit a Gaussian to the data might work better.
- Replacing probability optimization with a learned heuristic (combination of probabilities, resources spent, strength of units or other factors).
- Employing the *reactive* ASP paradigm [7], which can be viewed as an extension to ASP, ideal for dynamically changing domains like RTS games. Online ASP solvers like oClingo[3] process an initial logic program, compute the answer sets, but do not terminate. Instead they keep running as a server applications and wait for further knowledge updates (observations), while preserving and reusing all the previous computations. With this speedup, an artificial RTS player would potentially be able process new observations and generate predictions with much higher frequency.

[3] `http://www.cs.uni-potsdam.de/oclingo/`

## References

[1] Deep Thought project. Available at http://deepthought.23inch.de/. Accessed: 2013-08-30.

[2] Chitta Baral. *Knowledge representation, reasoning and declarative problem solving*. Cambridge University Press, 2003.

[3] Michael Buro. Call for AI research in RTS games. In *Proceedings of the 4th Workshop on Challenges in Game AI*, pages 139–142, 2004.

[4] David Churchill and Michael Buro. Portfolio greedy search and simulation for large-scale combat in StarCraft. In *Proceedings of Computational Intelligence in Games (CIG)*, pages 1–8, 2013.

[5] Ethan W Dereszynski, Jesse Hostetler, Alan Fern, Thomas G Dietterich, Thao-Trang Hoang, and Mark Udarbe. Learning probabilistic behavior models in real-time strategy games. In *Proceedings of AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 20–25, 2011.

[6] Steve Dworschak, Susanne Grell, Victoria J Nikiforova, Torsten Schaub, and Joachim Selbig. Modeling biological networks by action languages via Answer Set Programming. *Constraints*, 13(1-2):21–65, 2008.

[7] Martin Gebser, Torsten Grote, Roland Kaminski, and Torsten Schaub. Reactive answer set programming. In *Proceedings of Logic Programming and Nonmonotonic Reasoning (LPNMR)*, pages 54–66. Springer, 2011.

[8] Martin Gebser, Carito Guziolowski, Mihail Ivanchev, Torsten Schaub, Anne Siegel, Sven Thiele, and Philippe Veber. Repair and prediction (under inconsistency) in large biological networks with Answer Set Programming. In *Proceedings of the Twelfth International Conference on Principles of Knowledge Representation and Reasoning (KR)*, pages 497–507, 2010.

[9] Martin Gebser, Roland Kaminski, Benjamin Kaufmann, Max Ostrowski, Torsten Schaub, and Sven Thiele. A users guide to gringo, clasp, clingo, and iclingo, 2008. Available at: http://potassco.sourceforge.net/.

[10] Christopher W Geib and Robert P Goldman. A probabilistic plan recognition algorithm based on plan tree grammars. *Artificial Intelligence*, 173(11):1101–1132, 2009.

[11] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In *Logic Programming: Proceedings of Fifth International Conference and Symposium*, volume 88, pages 1070–1080. MIT Press, Cambridge MA, 1988.

[12] Michael Gelfond and Vladimir Lifschitz. Classical negation in logic programs and disjunctive databases. *New generation computing*, 9(3-4):365–385, 1991.

[13] Froduald Kabanza, Philipe Bellefeuille, Francis Bisson, Abder Rezak Benaskeur, and Hengameh Irandoust. Opponent behaviour recognition for real-time strategy games. In *AAAI Workshop on Plan, Activity, and Intent Recognition*, 2010.

[14] Maximilian Möller, Marius Schneider, Martin Wegner, and Torsten Schaub. Centurio, a general game player: Parallel, Java- and ASP-based. *KI-Künstliche Intelligenz*, 25(1):17–24, 2011.

[15] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game AI research and competition in StarCraft. *IEEE Transactions on Computational Intelligence and AI in Games*, 5:293–311, 2013.

[16] Marius Stanescu, Sergio Poo Hernandez, Graham Erickson, Russel Greiner, and Michael Buro. Predicting army combat outcomes in StarCraft. In *Ninth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2013.

[17] Gabriel Synnaeve and Pierre Bessiere. A bayesian model for opening prediction in RTS games with application to StarCraft. In *IEEE Conference on Computational Intelligence and Games (CIG)*, pages 281–288, 2011.

[18] Gabriel Synnaeve, Pierre Bessiere, et al. A dataset for StarCraft AI & an example of armies clustering. In *AIIDE Workshop on AI in Adversarial Real-Time Games*, volume 2012, pages 25–30, 2012.

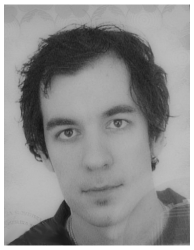
[19] Michael Thielscher. Answer Set Programming for single-player games in general game playing. In *Logic Programming*, pages 327–341. Springer, 2009.

[20] Michal Čertický. Implementing a wall-in building placement in StarCraft with declarative programming. *arXiv preprint arXiv:1306.4460*, 2013.

[21] Ben George Weber and Michael Mateas. A data mining approach to strategy prediction. In *Proceedings of IEEE Symposium on Computational Intelligence in Games (CIG)*, pages 140–147, 2009.

[22] Ben George Weber, Michael Mateas, and Arnav Jhala. A particle model for state estimation in real-time strategy games. In *Proceedings of AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE)*, pages 103–108, 2011.

**Marius Stanescu** is currently a PhD student with the GAMES group in the Department of Computing Science at University of Alberta. He completed his MSc in Artificial Intelligence at University of Edinburgh in 2011, and was a researcher at the Center of Nanosciences for Renewable & Alternative Energy Sources of University of Bucharest in 2012. Since 2013, he is helping organizing the AIIDE StarCraft Competition. Marius' main areas of research interest are machine learning, artificial intelligence, and RTS games.



**Michal Čertický** is a researcher at Agent Technology Center of Czech Technical University in Prague. Prior to his current position, Michal got his Ph.D. and RNDr. degrees in Computer Science at the Department of Applied Informatics of Comenius University in Bratislava, as a member of the Knowledge Representation Group (KRG). Since 2011, he has been organizing a StarCraft tournament of artificial intelligence bots, SSCAI.