Towards Procedural Strategy Game Generation: Evolving Complementary Unit Types

Tobias Mahlmann, Julian Togelius, Georgios N. Yannakakis

IT University of Copenhagen, Rued Langaards Vej 7, 2300 Copenhagen, Denmark
{tmah, juto, yannakakis}@itu.dk

Abstract. The Strategy Game Description Game Language (SGDL) is intended to become a complete description of all aspects of strategy games, including rules, parameters, scenarios, maps, and unit types. One of the main envisioned uses of SGDL, in combination with an evolutionary algorithm and appropriate fitness functions, is to allow the generation of complete new strategy games or variations of old ones. This paper presents a first version of SGDL, capable of describing unit types and their properties, together with plans for how it will be extended to other sub-domains of strategy games. As a proof of the viability of the idea and implementation, an experiment is presented where unit types are evolved so as to generate complementary properties. A fitness function based on Monte Carlo simulation of gameplay is devised to test complementarity.

1 Introduction

Strategy games are one of the most enduring and consistently popular game genres, and have been around in one form or another for hundreds of years. This genre of games is famous for being one of the most cerebral; world championship tournaments exist for several such games. Meanwhile, the long learning curve and strong skill differentiation usually leads dedicated strategy game players to devote immense amounts of time to playing those games. Strategy games designed to mimic real life scenarios are commonly used for training and simulation. At the same time, the design, development and balancing of a modern digital strategy game such as the latest instalments of the *Civilization* or *Starcraft* series is very labour-intensive and therefore expensive. Automating the design, development and tuning of strategy games would therefore be highly desirable.

The field of procedural content generation (PCG) is devoted to algorithms that automatically create various types of game content. While isolated examples of PCG in games date three decades back, and the *SpeedTree* software is commonly used for creating vegetation in commercial games, it is very rare to see PCG used for "necessary" content such as levels and mechanics rather than just for peripheral, "optional" content such as textures and collectable items in published games. Further, most PCG algorithms in published games are not *controllable*, simply generating random content within bounds.

Recently, the term *search-based procedural content generation* (SBPCG) was proposed for PCG algorithms that build on global stochastic search algorithms

(such as evolutionary computation) and fitness functions designed to measure the quality of game content [14]. Examples of this approach include the evolution of platform game levels [7], of racing game tracks [11] and the distributed evolution of weapons in a space shooter game [3].

But what about the most fundamental aspects of games: their rules and the mechanics they imply? A language for describing the rules of a game (and possibly other aspects) is known as a game description language (GDL). Several GDLs have been proposed for different purposes. The Stanford GDL, created for the general game playing competition, is a relatively genre-independent language yet limited to perfect information games with discrete state space [6]. Based on first-order logic, the Stanford GDL tends to be rather lengthy: a description of Tic-Tac-Toe is approximately 3 pages long. Browne's *Ludi* GDL trades generality for conciseness — by limiting itself to two-player board games with restrictions to pieces and boards, it allows Tic-Tac-Toe to be described in 6 lines [1]. An interesting GDL variation is Smith's and Mateas' *Ludocore*, which expresses 2D arcade games using logic programming [10]. To the best of our knowledge, a GDL suited for describing strategy games has not yet been introduced.

Measuring the quality of a rule set appears to be a rather challenging task. It is not clear what sort of rule set qualities one would like to incorporate within a utility function. Meaningfulness and accessibility of a rule set are two options among many. Some recent work has focused on the design of fitness functions based on empirical measures of player experience, but this has not yet been attempted for strategy games [16, 7, 17]. While a human can to some extent judge the quality of level or character design of a game by just looking at it, you need to play a game to judge the quality of its rule set; it stands to reason that the same should be true for algorithms. Therefore, functions that accurately measure the quality of game rules are likely to be simulation-based (according to the classification presented in [14]), meaning that the functions build on the game being played by an algorithm. Browne measured the quality of board games using a number of custom-defined measurements [1], most of them simulation-based. Togelius and Schmidhuber proposed a learnability-based fitness function, where the entertainment values of 2D predator-prey games are estimated by how they can be learnt by an algorithm [13]. Salge and Mahlmann evaluated simple strategy game battles using the information-theoretic concept of *relevant information* to determine the amount of information necessary to play well [9].

The only study of search-based PCG applied to strategy games that we are aware of is focused on evolving maps for such games [12]. While the approach taken in that work could conceivably form a component of a system for generating complete strategy games, it only addresses one aspect of strategy games.

In this paper, we investigate how PCG techniques could be potentially applied to all aspects of strategy games. Our aim is to create games that have the potential for *deep gameplay*: games where a multitude strategies are possible, and that reward exploring ever more sophisticated strategies . We advocate a search-based approach based on a customized game description language and simulation-based fitness functions. The contributions of this paper are as follows:

1. A plan for a strategy game description language (SGDL) for all aspects of strategy games;
2. some examples of SGDL in the domain of unit types;
3. a simulation-based fitness function for measuring the complementarity of units; and
4. an experiment where the SGDL fragment and the fitness function are combined to evolve sets of unit types.

2 The Strategy Game Description Language

This section presents the main elements of the strategy game description language including our design criteria and SGDL's overall structure. Our design criteria for SGDL are that it should be:

- **complete**: able to model all, or almost all, aspects of a rich variety of strategy games, including some published and successful games.
- **evolvable**: be easy to search within. One of the main implications of this property is that the language should have a high *locality*, meaning that that similar descriptions (genotypes) in general give rise to games (phenotypes) with similar fitness values.
- **human-readable**: it should be easy for a human to understand and edit SGDL strings (genotypes), at least on a micro-level.

2.1 Overall structure

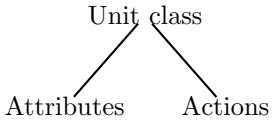
A strategy game can be decomposed into three layers:

1. The *mechanics* layer. This layer determines the fundamental rules of the game, such as what an attack action is, what it means to win the game and in what type of game environment units are placed on (e.g. on a 2D grid).
2. The *ontology* layer. This layer specifies the types of key elements that may exist in the game (e.g. rivers, mountains, tanks and factories) as well as their properties (e.g. mountains have movement cost 5 for ground units).
3. The *instance* layer. The setup of an individual match, campaign or battle are specified within this layer: the layout of the map, initial placement of units, and any particular conditions that might apply (e.g. there is no fog of war and the battle is lost if enemy survives after 100 turns).

We propose to describe at least the mechanics and ontology layers using a tree-based representation, similar to the most common representations used in genetic programming [8]. In the following, we reveal the basic structure of SGDL using a strategy game *unit* as an example. Units are the most common game elements a player can interact with in a strategy game. Units can be compared to the pieces of a board game such as Chess. Furthermore, the challenge of the game is increased since units usually belong to different classes which provides them with dissimilar abilities and properties. A game often uses its theme to

make it more intuitive what the differences are, e.g. tanks and airplanes are intuitively seen as objects with different properties.

In SGDL, an object class consists of three components: (1) A unique alphanumerical name/identifier ; (2) a set of attributes that are either numerical, alphanumerical or boolean; and (3) a set of actions that consist of conditions and effects. All these key components can be seen as nodes in a structural tree.



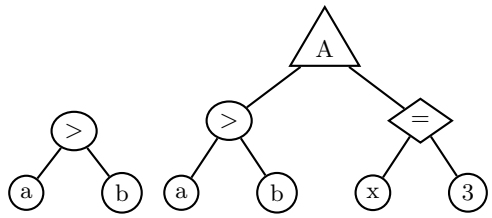
Since the left *Attributes* subtree of a unit only consists of leaf nodes pairing an alphanumerical identifier and a value, we'd like to focus on the *Actions* subtree in this paper. Before we discuss this, we would like to give a short overview of the structure. Our language currently supports the following nodes:

1. *Actions* (triangle shape) are container nodes, combining *conditions* and *consequences*. If an action is invoked, all conditions are tested against the invocation parameters. If all return true, all consequences are executed.
2. *Comparators* (oval shape) combine their children's outputs and return a boolean value to their parent. There are also two special types for this class:
   (a) *Object reference or parameter nodes* can be used to refer to an instance of a class that was passed into the current invocation of the action.
   (b) *Special function nodes* take a parsable string and one or many child nodes' outputs to perform operations such as accessing the game's map.
3. *Operators* (diamond shaped) have different behaviours depending on their value: set operators like = or ! combine the value of their right child with the operator and assign the outcome to their left child. Mathematical operators $(+,-,*,/)$ behave like comparator nodes, but return numerical values instead.
4. *Constants* (circular shaped) are leaf nodes and may contain contain a constant alpha- or numerical value.

Actions are sets of *if ... then* rules for each unit the player may choose from. While the instance layer should define *when* a player may choose an action, the mechanics layer defines the conditions and outcome of an action. We describe how to express conditions and consequences below.

The representation of conditions mirror the parsed form of mathematical tests, e.g. the simple mathematical formula $a > b$ can be represented as seen in Figure 1(a). Consequences can also be seen as mathematical operations, e.g. variable assignments. In Figure 1(b) we show how to represent the action A with the condition $a > b$ and the consequence let $x = 3$.

While conditions like *if attribute x can be assigned 3 then ...* would theoretically be possible, we exclude these cases for now as they would generate unwanted side effects on condition testing; it would be possible to alter the game state by testing actions that are prevented from happen by other conditions. One way to solve this would be to duplicate all referenced values during testing.



(a) A condition (b) A condition combined with a consequence

Fig. 1. Two elementary bits of SGDL

On the other hand, consequences may also trigger a sequence of follow-up actions. Figure 2(a) should be read as: *if a is greater than b then let x be 3 and if additionally c equals d then let y be 5*.

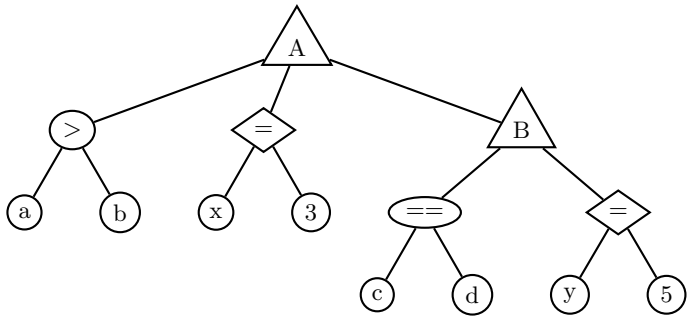
Figure 2(b) depicts the action *Go North*, which is used by units capable of moving on a two-dimensional quadratic map: The action is possible if the output of the special $\_MAP$ node equals *null*. That node has two input parameters: the x attribute of the first object passed into the action and its y attribute subtracted by one. The consequence is that the first object's y attribute is in fact subtracted by one. While the other movement actions are modelled similarly, the attack actions are more complex and involve a target object.

3 Evolving complementary unit types

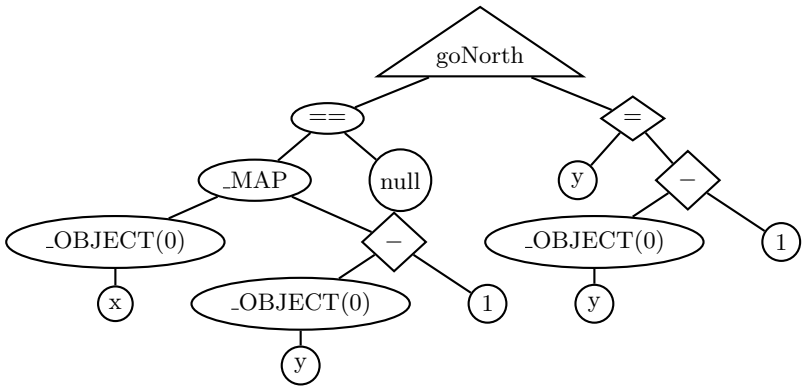
As a proof of concept, and of implementation, we conducted an experiment where we evolved complimentary unit types. A set of unit types is complimentary when they have different strengths, so that each of them is better than the others in some respect; and when combining the units is in general better to have a balanced set than having the equivalent number of units of only one type. Many strategy games include the unit types infantry, artillery and helicopter (or something similar), where each unit type has unique strengths and weaknesses so that a successful strategy depends on using them in combination. To emphasize how important this aspect is for commercially successful strategy games, we would like to point out that the closed beta testing for Blizzard's *Starcraft II* ran over five months with 40.000 players participating [15].

3.1 Method

We designed a very simple strategy game for our studies. The game takes place on a 10×10 regular quadratic map. Each player has three units, and those start evenly spaced out on opposite sides of the map. Each turn, one player can move



(a) An example of SGDL



(b) A simple *Go North* example

Fig. 2. Tree demonstrating a more complex example and an action used in our test bed.

or attack with one of his units[1]. A unit can move one step north, south, east, west, or attack one of the enemy units. Units cannot move outside the map or attack a unit which is not within their range.

Each unit type has seven attributes: *health* (range $[0, 100]$), *ammunition* ($[0, 100]$), three *attack* values ($[0, 100]$) and both maximum and minimum attack range ($[0, 6]$). The attack values determine the damage that can be done by one shot on each of the enemy unit types. This means that to define a complete set of unit types, 21 values need to be specified (seven per unit class; three classes in the game). Additionally, the value range restrictions are dropped after the genome was created, letting the genetic algorithm change values arbitrarily.

In order to search this 21-dimensional space for sets of complementary unit types, we employed a $\mu + \lambda$ evolution strategy with $\mu = \lambda = 50$. For simplicity, neither crossover nor self-adaptation was used. The mutation operator added

[1] We realize that this design choice, which was due to computational efficiency considerations, in some ways makes our game resemble a traditional board game rather than a strategy game; this is discussed further in Section 4.

Gaussian noise with $\mu = 0, \sigma = 0.1$ to all values in the genome. The gene values were normalized to real values in the range $[0, 1]$.

Designing a fitness function capable of accurately measuring unit type set complementarity proved to be a challenge. As a prerequisite for a simulation-based fitness function we need to be able to play out battles automatically. This was achieved through Monte Carlo tree search (MCTS) with upper confidence bounds applied to trees (UCT) [5]. When playing a game, each player has to choose between a minimum of 2 and maximum of 21 available actions at each turn, depending on the number of units at his disposal, how many targets are in range and where units are placed. Action selection works by taking each action in a copy of the game engine, and do 100 rollouts of random action sequences; the action with the best average outcome (defined as difference in total health between the two players) is chosen. The initial rollouts are 5 turns long. If the difference between the outcomes of different actions is not significant with the 5-turn rollouts, 20-turn rollouts are performed.

Building on this foundation of automated gameplay, the actual fitness function was implemented as follows: six battles were played for each unit type set. Balanced unit sets (denoted ABC) played against unbalanced sets with units of only one type (AAA, BBB and CCC). Three games where played where the balanced unit set started the game, and three games where the unbalanced set started. The fitness was defined as the minimum fitness achieved by the balanced set in any of the six games. To minimize noise, the fitness calculation was averaged over 200 trials (t). This led to a computationally expensive fitness function, taking more than a minute of computation on cluster node (a *Core 2 Duo* with 2.4GHz and 2GB RAM). More formally, the fitness of a genome is:

$$F := min(\frac{\sum^t a_1 + a_2}{t}, min(\frac{\sum^t b_1 + b_2}{t}, \frac{\sum^t c_1 + c_2}{t}))$$

where $a_1, a_2, b_1, b_2, c_1, c_2$ are defined as 1 if the player with the balanced set has won against the according non-balanced set, or 0 otherwise, and $t = 200$.

3.2 Results

We ran several evolutionary runs of 100 generations, each one taking several days on a cluster of six computers. A graph of a typical run is depicted in Figure 3. An early and steady increase in the maximum fitness value suggests that the fitness function can be optimized effectively using a standard evolutionary setup; the maximum fitness reaches 0.86 in the run illustrated here.

3.3 Analysis of evolved unit type sets

A key research question in this experiment is whether the fitness function accurately captures the desired property of complementarity, and whether the highly fit unit type sets are more interesting to play than poorly fit sets. To shed some light on this, we analyse a few evolved unit type sets.

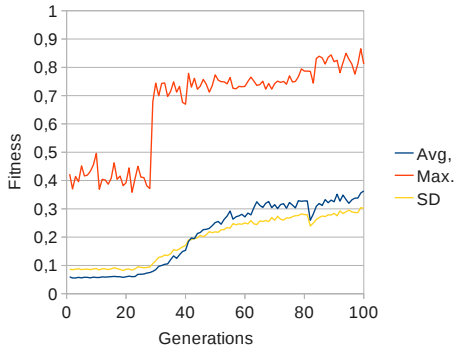


Fig. 3. Average and maximum fitness during an evolutionary run.

Type	Health	Ammo	Attack 1	Attack 2	Attack 3	Min range	Max range
A	53.0	33.0	60.0	20.0	92.0	10.0	**0.0**
B	82.0	78.0	85.0	60.0	62.0	0.0	23.0
C	39.0	45.0	37.0	100.0	12.0	0.0	**0.0**

Table 1. A unit type set with fitness 0.0.

Table 1 presents one unit type set with fitness of 0.0. We can see that that this particular set contains two basically non-functional unit types: the A and C unit types are unable to shoot given that their shooting range is zero. While games against AAA and CCC will always end in favour of ABC, ABC will never win against BBB. Even though ABC contains one functional unit and may even kill one unit of BBB, it will always be eliminated by the second unit of BBB. Therefore there exists a dominant combination that always wins over all other combinations, making this configuration very uninteresting to play.

Table 2 presents a set with fitness of 0.24, which is a mediocre score. While all three unit types appear to be functional and have different strengths and weaknesses, this configuration does not perform very well. We believe that this might be due to the observation, that all three types have very similar minimum and maximum ranges. In conjunction with the alternating turn order it may become a losing proposition to ever engage an enemy unit. The unit that moves in range first will inevitably be the first one to take damage since the enemy moves next. As our MCTS-based player will avoid most such moves, most games will be counted as unplayable after a turn limit of 100. The positive fitness is probably because some games are won by one party or another by pure chance.

Table 3 presents the top-scoring individual found during one evolutionary run described above. The unit types' attack values are rather complementary — each unit type vulnerable against at least another type. We see also see that type C has more health than the other types. Type A and B can be seen as support units, while type C is more of a general purpose combat unit. Units of

Type	Health	Ammo	Attack 1	Attack 2	Attack 3	Min range	Max range
A	46.0	69.0	61.0	71.0	71.0	2.0	5.0
B	6.0	43.0	22.0	90.0	22.0	3.0	5.0
C	36.0	82.0	40.0	47.0	6.0	2.0	4.0

Table 2. A unit type set with fitness 0.24.

Type	Health	Ammo	Attack 1	Attack 2	Attack 3	Min range	Max range
A	6.0	82.0	39.0	2.0	67.0	0.0	3.0
B	4.0	31.0	92.0	79.0	3.0	1.0	5.0
C	64.0	79.0	94.0	1.0	90.0	0.0	2.0

Table 3. A unit type set with fitness 0.57.

type A and B can be killed with a single shot. Ammunition values are such that all units may shoot without any shortage of ammo.

4 Discussion

In this paper, we have introduced the first steps towards the automatic design and balancing of complete strategy games. We have described the basic structure of a strategy game description language, a simulation-based fitness function for complementary unit type sets and some initial results for evolving such sets.

While we can evolve highly fit unit type sets, games using the evolved units are not very interesting to play. This suggests either a flaw in our fitness function or an inherent inability of the simple test-bed game used in this paper to support deep gameplay. We will test our approach with more complex game environments which are closer to well-known strategy games: this would include more units, larger maps with terrain features, and ability for players to use all their units each turn, hoping to achieve more interesting gameplay.

In addition we aim to develop more sophisticated and reliable fitness functions. The *theory-driven* approach for the design of such functions would be to adopt principles of interestingness for board strategy games and build heuristics based on those principles (e.g. the player tension model [2] or the entertainment model [4] proposed by Iida et al.). An alternative *data-driven* approach would be to let human subjects play and rank a number of unit type sets on several dimensions, such as gameplay depth, challenge, accessibility etc. We could then develop a number of new fitness functions, and create a nonlinear model that estimates the various dimensions of player experience based on this collection of fitness functions [7]. Those fitness functions could then be used to guide the evolution of strategy game content for particular players and desired experiences.

As a long-term goal, we are interested in evolving all aspects of strategy games. This will require extending SGDL to describing e.g. game rules and terrain types. We aim to do this using the current tree-based structure.

References

1. Browne, C.: Automatic generation and evaluation of recombination games. Ph.D. thesis, Queensland University of Technology (2008)
2. Cincotti, A., Iida, H.: Outcome uncertainty and interestedness in game-playing: A case study using synchronized hex. New Mathematics and Natural Computation (NMNC) 2, 173–181 (07 2006)
3. Hastings, E., Guha, R., Stanley, K.O.: Evolving content in the galactic arms race video game. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG) (2009)
4. Iida, H., Takeshita, N., Yoshimura, J.: A metric for entertainment of boardgames: its implication for evolution of chess variants. In: Nakatsu, R., Hoshino, J. (eds.) IWEC2002 Proceedings. pp. 65–72. Kluwer (2003)
5. Kocsis, L., Szepesvári, C.: Bandit based monte-carlo planning. Machine Learning: ECML 2006 pp. 282–293 (2006)
6. Love, N., Hinrichs, T., Haley, D., Schkufza, E., Genesereth, M.: General Game Playing: Game Description Language Specification (2008)
7. Pedersen, C., Togelius, J., Yannakakis, G.N.: Modeling Player Experience for Content Creation. IEEE Transactions on Computational Intelligence and AI in Games 2(1), 54–67 (2010)
8. Poli, R., Langdon, W.B., McPhee, N.F.: A field guide to genetic programming. Published via `http://lulu.com` and freely available at `http://www.gp-field-guide.org.uk` (2008), http://www.gp-field-guide.org.uk, (With contributions by J. R. Koza)
9. Salge, C., Mahlmann, T.: Relevant information as a formalised approach to evaluate game mechanics. In: Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG) (2010)
10. Smith, A.M., Mateas, M.: Variations forever: Flexibly generating rulesets from a sculptable design space of mini-games. In: Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG) (2010)
11. Togelius, J., De Nardi, R., Lucas, S.M.: Towards automatic personalised content creation in racing games. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG) (2007)
12. Togelius, J., Preuss, M., Beume, N., Wessing, S., Hagelbäck, J., Yannakakis, G.N.: Multiobjective exploration of the starcraft map space. In: Proceedings of the IEEE Conference on Computational Intelligence and Games (CIG) (2010)
13. Togelius, J., Schmidhuber, J.: An experiment in automatic game design. In: Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG) (2008)
14. Togelius, J., Yannakakis, G.N., Stanley, K.O., Browne, C.: Search-based procedural content generation. In: Proceedings of EvoApplications. vol. 6024. Springer LNCS (2010)
15. Various: The starcraft wikia article about the starcraft ii beta test (Oct 2010), http://starcraft.wikia.com/wiki/StarCraft_II_beta
16. Yannakakis, G.N.: How to Model and Augment Player Satisfaction: A Review. In: Proceedings of the 1st Workshop on Child, Computer and Interaction. ACM Press, Chania, Crete (October 2008)
17. Yannakakis, G.N., Hallam, J.: Towards Optimizing Entertainment in Computer Games. Applied Artificial Intelligence 21, 933–971 (2007)