TRANSLATING NATURAL LANGUAGE TO THE GAME DESCRIPTION LANGUAGE

By

Alex Haig

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF SCIENCE

Major Subject: COMPUTER SCIENCE

Examining Committee:

Selmer Bringsjord, Thesis Adviser

Sergei Nirenburg, Member

Mei Si, Member

Rensselaer Polytechnic Institute
Troy, New York

July 2014
(For Graduation August 2014)

CONTENTS

LIST OF FIGURES iii

ABSTRACT iv

1. INTRODUCTION 1

2. THE GDL AND DOMAIN 3

2.1 General Game Playing Game Model 3

2.2 The Game Description Language 3

2.3 Domain 7

3. NATURAL LANGUAGE PROGRAMMING 10

3.1 Comparison to Other Systems 10

3.2 Why Dialog? 13

4. IMPLEMENTATION DETAILS 15

4.1 Dialog 15

4.2 Processing Input 16

4.2.1 Parsing 17

4.2.2 Analyzing Parsed Input 18

4.3 Intermediate Representation 18

4.4 Outputting the GDL code 19

4.5 Verification of the GDL code 20

5. FUTURE WORK 22

5.1 Back to Blokus 24

6. CONCLUSION 26

REFERENCES 27

APPENDICES

A. Generated Code for Tic-Tac-Toe 29

B. Semantic Grammar for Goal and Terminal Conditions 35

LIST OF FIGURES

4.1 Dialog Finite State Machine 15

4.2 Successful Validation of Generated GDL for Tic-Tac-Toe 20

ABSTRACT

This thesis presents work done on the problem of translating natural language descriptions of games into the Game Description Language (GDL) — a logic programming language used to describe games in the field of general game playing. This problem is interesting in that, if solved, it would open up general game playing to the general public and could eventually serve as part of a larger system allowing users to face computer opponents in almost any game they wished. The approach taken in this project was to develop a system that engages in an interactive dialog with the user to obtain the user's description of the game they desired to create. After the system obtains the game description, it is processed through various techniques such as semantic grammars, and then code created from a set of building blocks is produced. The current system presented herein is a successful proof-of-concept and is able to produce GDL code from natural-language descriptions of games in a domain consisting of Connect-4, Tic-Tac-Toe, and some basic variations thereof. While this domain may be small, the methods used hold promise for an expanded and more complete version of the system in the future.

1. INTRODUCTION

Imagine someone being bored at work (during a break of course). They look at their phone for a game to play, but nothing seems to be quite the exciting, intellectually fulfilling, abstract strategy game they really want to play. So they say to their phone, "I want to play a new game". "What kind of game?" it asks. In this case the user wants to play the game Blokus (they are not familiar with the game already). They say: "There are a bunch of pieces of different shapes and sizes that the players are trying to fill a board with". Then the system starts asking questions like: "What size is the board?", "How many players are there?" and "How many pieces are there?". The user would answer each of these questions, and then the system and user would begin a discussion of how each of the pieces are shaped. Eventually the system might ask "How do the players win?" and the user would respond "Whoever has the least number of squares left over at the end of the games wins". At this point the system would have to ask the user some questions to understand exactly what they meant; for example "What do you mean by square?". After some further back and forth, the system would eventually understand the user's game, and then could play the game with the user.

While such an interaction between man and machine is still very much a figment of imagination, various components of this system are actively under development — in fact one system aiming at allowing users to describe the game Pac-Man has been proposed in [1] and provided inspiration for this project.

The first component is the field of general game playing. Instead of creating systems that can play one game, for example, chess, in the case of IBM's Deep Blue, general game playing seeks to create systems that can play any number of games given their descriptions [2, 3]. In the scenario above, a general game playing system would play the game with the user once it had been described.

The second component, and the focus of this project, is taking the natural language input of the user and translating it into the description language used by the general game playing system — in this case the Game Description Language

(GDL). The development of this system will make it easier for those in the general game playing community to create games, and eventually perfecting this process would allow for the fast and easy creation of games without any knowledge of logic or programming at all. Ideally, this system would cover the same domain of games as the GDL itself, but at the very least it would have to be able to describe many well-known abstract strategy games and the combinations and variations thereof. The purpose of this project is to provide a proof of concept for such a natural-language-to-GDL translation system.

2. THE GDL AND DOMAIN

2.1 General Game Playing Game Model

In general game playing, games are modeled as a finite state graph with a unique initial state and multiple terminal states [2, 3]. Games are played by starting at the initial state, and then traversing the graph through a series of steps until a terminal state is reached. At each step, all players are required to take one of the available actions that is also legal in that state and the actions taken by the players determine the next state. To accommodate for turn-based games, the action taken may be to do nothing — a "noop" action.

One issue with this representation of games is that many games will have a large number of states. Even Connect-4 — a relatively simple game — has 4,531,985,219,092 different board positions [4]. Therefore, representing games as a graph is not feasible, and a more compact representation is needed [3, 2]. This is the purpose of the Game Description Language (GDL).

2.2 The Game Description Language

The GDL is a declarative, logic programming language derived from Datalog that compactly describes a finite state machine representing a game [2, 5]. Like other logic programming languages, GDL programs are written as a set of relations and a set of rules made up of a head – a relation constant with n terms — and a body — a conjunction of literals. Unlike some other logic programming languages where variables are marked by uppercase letters, in the GDL variables begin with "?". For the purposes of representing game concepts, the following special relations are predefined in the GDL [2, 3]:

base(p): This means that proposition p — an n-ary relation with entities as terms — is a base proposition in the game. The set of base propositions must contain all the propositions that can be formed from the game's entities and relations.

```
(<= (base (cell ?col ?row none none))
    (is_column ?col)
    (is_row ?row))
```

Above is an example of a **base** statement for the game Tic-Tac-Toe (all examples for the relations in this section will be for Tic-Tac-Toe) that says that for every pair of numbers (?col, ?row), if the board has a column numbered ?col and a row numbered ?row then (cell ?col ?row none none) is a base proposition.

init(*p*): This means that proposition ***p*** is true in the initial state. The initial state of Tic-Tac-Toe could be describe as follows:

```
(init (cell 1 1 none none))
(init (cell 1 2 none none))
            ⋮
(init (cell 3 3 none none))
(init (control player1))
```

Initially the board is empty, so all the cells are set to have nothing in them, and it is player1's turn so player1 is given "control" — a term that is often used to indicate who's turn it is.

true(*p*): This means that proposition ***p*** is true in the current state. **true** is used as an input to the **legal**, **goal**, **terminal**, and **next** relations, and therefore cannot be in the head of any rule.

role(*a*): This defines ***a*** as a role (player) in the game. Tic-Tac-Toe has two players so the **role** statements will look as follows:

```
(role player1)
(role player2)
```

input(*r,a*): This means that ***a*** is one of the possible actions that role ***r*** may take. Whether or not action ***a*** can be taken in the current state is dependent on the **legal** relation.

```
(<= (input ?player (place ?occupant ?col ?row))
    (role ?player)
    (is_column ?col)
    (is_row ?row)
    (owns ?player ?occupant))
```

The above code is an example **input** statement that says that an available action for a player to take is to place a piece that they own in a cell designated by a column and a row.

legal(*r,a*): This means role ***r*** may take action ***a*** in the current state. **legal** must be defined in terms of the **true** relation. The Tic-Tac-Toe rule that players can only mark open squares would be represented as follows:

```
(<= (legal ?player (place ?occupant ?col ?row))
    (owns ?player ?occupant)
    (true (control ?player))
    (open square ?col ?row))
```

The above code says: if in the current state it is the player's turn, the piece they want to place is theirs, and the square they want to place the piece in is open, then it is legal for them to place the piece in that square.

does(*r,a*): This means role ***r*** takes action ***a*** in the current state. **does** is used as input for the **next** relation as follows:

```
(<= (next (cell ?col ?row ?player ?occupant))
    (does ?player (place ?occupant ?col ?row)))
```

next(*p*): This means proposition ***p*** will be true in the next state. **next** must be defined in terms of the **true** and/or **does** relations.

```
(<= (next (control player1))
    (true (control player2)))

(<= (next (cell ?col ?row ?player ?occupant))
    (does ?player (place ?occupant ?col ?row)))
```

Above are two examples of the **next** relation. In the first, it is stated that if in the current state player2 has control (it is player2's turn) then in the next state player1 will have control. The second states that if the player places a piece in some square (indicated by column and row numbers), then in the next state there will be a piece in that square. It should be noted that everything about the next state must be defined, so it must be explicitly stated that if a player does not place a piece in a square, then that square will be empty in the next state. This could be done as follows:

```
(<= (next (cell ?col ?row ?cell_player ?occupant))
    (true (cell ?col ?row ?cell_player ?occupant))
    (does ?move_player (place ?occupant ?dest_col ?dest_row))
    (distinct_cells ?col ?row ?dest_col ?dest_row))
```

goal(*r*,*n*): This means the current state has value ***n*** for role ***r***. **goal** must be defined in terms of the **true** relation.

```
(<= (goal ?player 100)
    (win ?player))
```

This is a simple example where the state where a player wins has value 100 for that player. It may appear that this has not been defined in terms of the

true relation; however, this is because **true** is embedded inside of the user defined **win** relation.

terminal This means that the current state is a terminal state. **terminal** must be defined in terms of the **true** relation.

```
(<= terminal
    game_end)

(<= game_end
    (win ?player))
```

This is another simple example showing that a state where the player wins is also a terminal state. Once again, the **true** statement is embedded inside the **win** relation.

Every GDL program must provide complete definitions for all of the above relations with the exception of **true**, which is used within the definitions of the relations specified. Another important restriction is that any variable in the head of a rule or a negative literal in the body of a rule must also be present in a positive literal in the body of a rule. This is to prevent infinitely large game models [5]. A complete list of restrictions on programs written in the GDL can be found in [2] and [5].

2.3 Domain

Apart from the restrictions on how games are written in the GDL, there are some other restrictions on what is considered a well-formed game in general game playing. Four explicit restrictions are as follows [5]:

Termination: All infinite sequences of legal moves starting from the initial state must reach a terminal state after a finite number of steps. This just means that a game cannot potentially go on forever. For example, in chess the game is a draw if the same position is repeated three times. If this was not the case

and players were able to continually move their pieces back and forth between positions forever, then chess would not be considered a well-formed game by general game playing standards. One way this issue is often handled is by simply imposing a limit on the number of turns a game may have so that no matter what, once a certain number of steps have passed the game will end.

Playability: Every role must have at least one legal move in every non-terminal state. According to the game model every player must make a move at every step. This requirement ensures that this is possible.

Monotonicity: Every role has one goal value in every state reachable from the initial state, and these goal values are non-decreasing.

Winnability: For every role, there is a sequence of moves made by some subset of all the roles that leads to a terminal state where that role's goal value is maximal. In abbreviated terms, this means that it must be possible for every player to win the game. However, it may be the case that in order for a player to win, an opponent must be cooperative and make moves that allow that player to win.

Some other implicit restrictions on the games due to the way the GDL is defined are that games must have perfect information and must be deterministic (however these issues are solved in GDL-II which is not considered here) [2, 5].

These restrictions leave an expansive domain of games for users to describe and play. Therefore, the domain that this project hopes to cover for now is a subset of the domain GDL covers. This custom domain is created by placing the following extra restrictions on the permitted games:

1. The game must be played on a rectangular grid of squares.
2. The game must have some form of piece(s). However, the pieces do not need to exist as a physical object — an X in Tic-Tac-Toe could be considered a piece.
3. Every square can hold at most one piece.

In its current stage as a proof of concept, this project actually covers a much smaller domain limited to only Connect-4, Tic-Tac-Toe, and some basic variations thereof.

3. NATURAL LANGUAGE PROGRAMMING

Now that the GDL has been well-defined, it can be seen that translating natural language to the GDL is a task in natural language programming within the specific domain of general game playing. It should be noted that "domain", as it is used here, is referring to the subset of English that must be handled by the system rather than the set of games the system is able to produce. Currently no other work has been done on a natural language programming system in this domain, however this project uses many of the same techniques implemented by natural language programming systems in other domains [6, 7].

3.1 Comparison to Other Systems

The biggest difference between this system and other natural language programming systems is the domain in terms of both content and size. As stated previously, no other work has been done in natural language programming for general game playing, and much of the novelty of this system comes from looking at this new domain. In terms of size, many other natural language programming applications focus on more general programming, and therefore the subset of English required for this project is much smaller than what is needed for those applications. One application domain that is comparable in size is natural language interfaces to databases when applied to one particular area. Still, natural language interfaces designed to be used in many different database applications have a larger domain than this project [6]. Having a smaller domain means that while many of the techniques used in larger domains will work in this domain, some of the issues these larger domains face will be mitigated. For example, the larger the domain, the worse ambiguity will be because there are more possible meanings for words and more contexts for those words to take place in. Also, some techniques that are not feasible in large domains, such as semantic grammars, are an option for this project because of the smaller domain.

Another difference between this system and some other natural language pro-

gramming systems is the intermediate representation. While there are a wide variety of different types of intermediate representations, one common method is to use natural language understanding to translate the input into a formal representation [6]. Since GDL is a logic programming language this may seem like an intuitive approach; however, GDL does not have the same constructs — such as existential and universal quantification — that make this approach viable for First Order Logic (FOL) which is what is most commonly used for such a logic-based approach [7]. This is not to say that the input could not be first translated into FOL and then translated into the GDL, however the intermediate representation used in this project for now — discussed in Section 4.3 — was chosen due to its simplicity.

Another distinguishing aspect of this system is the use of "active" English rather than "passive" English as described in [8]. Systems that use passive English are those where the language used for programming looks like English but still requires some programming knowledge or training with the language in order to be used. One prominent example of passive English languages are controlled natural languages such as Attempto Controlled English (ACE) [9, 10]. ACE looks and can be read like natural English, but it is in fact a formally defined language that requires training in order to be used. Examples of some systems that use passive English can be found in [11], [12], and [13].

In contrast, systems that use active English allow for programming by using a subset of English without any required programming knowledge or training. In these systems, the English input accepted is not constrained by the design of the system but by limitations in its implementation. One example is the natural language interfaces to databases found in [6]. These systems may not always understand the user's query, however they are designed so that the user can query the system in whatever way is natural for them without requiring training. Other systems that use active English can be seen in [14] and [15].

A system using a passive English language like ACE could be quite effective at describing games, and would likely be easier to translate into GDL code since it is formally defined and removes significant roadblocks such as ambiguity [9]. However, since the ultimate goal for this project is that people without training or program-

ming experience are able to create games, the project must focus on using active English.

Another technique used in this project that are often seen in other natural language programming applications is semantic grammars [6, 7]. Semantic grammars are similar to syntactic grammars except that instead of using syntactic concepts, they use semantic ones. For example, a syntactic parse of the sentence "The game ends if the board is full" using the Stanford Parser would look as follows [16]:

```
(ROOT
  (S
    (NP (DT The) (NN game))
    (VP (VBZ ends)
      (SBAR (IN if)
        (S
          (NP (DT the) (NN board))
          (VP (VBZ is)
            (ADJP (JJ full)))))))))
```

This parse shows syntactic structure of the sentence. On the other hand, a semantic parse of the same sentence using one of the semantic grammars designed for this project would look as follows:

```
(S
  (RESULT (ENTITY (TEMPORAL game)) (ACTION end))
  (IF if)
  (COND (STATE (ENTITY (BOARD_PART board)) (STATE full))))
```

In this case the parse shows the semantic structure of the sentence.

Semantic grammars are used because they get directly at the meaning of a sentence without having to worry about the syntax. A downside to semantic grammars is that they are domain-specific, but since the domain of this project is relatively small, they have worked well parsing the more complex inputs [7].

Another common technique that this system uses is dialog, and the reasons for using dialog will be discussed in the next section [6, 7].

3.2 Why Dialog?

There are three main reasons why interactive dialog was used to obtain the description of the game rather than static text.

First and foremost is the issue of disambiguation. Ambiguity is one of the biggest problems facing any natural language system and this system is no exception. For example, if in the description of a game the user makes a statement such as "player 1 beats player 2 if they have no pieces on the board" it is difficult to tell if the user meant that player 1 wins if either player 1 has no pieces on the board, player 2 has no pieces on the board, or neither player has pieces on the board. A system with no capacity for dialog would be forced to make a best guess, but with dialog the system could ask the user for clarification and produce the result desired by the user. In the current system this form of disambiguation is very limited, but choosing to use dialog opens up the potential for more complex disambiguation in the future.

The second reason to use dialog is to make sure that all of the information required for a valid and complete game description is provided by the user. The user should not need to know anything about the GDL and its requirements, and a game description produced by the user could easily be missing information required to write the GDL code even if it would make perfect sense to a human reader. A dialog prevents this from happening because the system can explicitly ask the user for any piece of information still needed to create a valid and complete game description.

Finally, dialog can make processing the user input easier. Whether the dialog system is a simple finite state machine — as it currently is — or a more advanced frame-based system, the dialog can be broken up into different states based on what component of the game is being discussed. For example, one state may be talking about the board setup while another state may be talking about the game pieces. Breaking up the input into these states makes it easier for the system to determine

what GDL code needs to be written since only some of the available code building blocks will be associated with that state. Furthermore, not all of these states require the same level of sophistication in the input processing, so the input processing at each state can be tailor made for the types of inputs expected in that state. This is discussed further Section 4.2.

4. IMPLEMENTATION DETAILS

This section covers the different components of the system and presents some details about how they are implemented. An overview of the system pipeline is as follows: First the system queries the user for some information about the game in the form of a text-based dialog. Then, the user input is processed based on the current state of the dialog and stored in an intermediate representation. This intermediate representation determines which blocks of GDL code are needed and how to fill variable parts of the code. All of these blocks are then written to a file. Finally, the code is verified for correctness. All code for this project is open source and can be found online[1].

4.1 Dialog

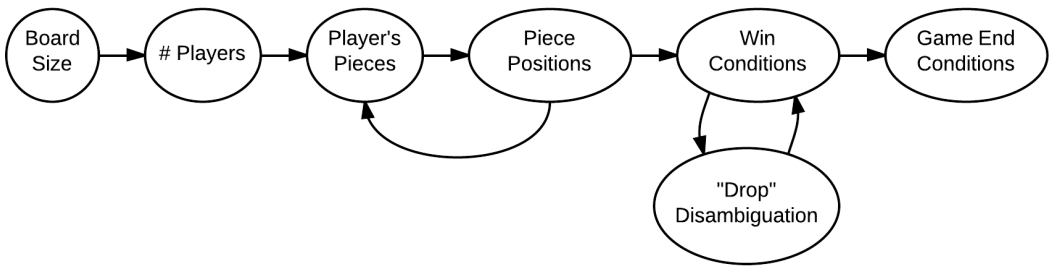

Figure 4.1: Dialog Finite State Machine

Currently the dialog system used in this project can be represented as the simple finite state machine seen in Figure 4.1. Since the components needed to fully define a game are known ahead of time, the system can just query the user for each of the components. The following is a sample dialog where the user is describing the game Tic-Tac-Toe:

[1]https://github.com/katadh/ngdl (Retrieved June, 23, 2014)

SYSTEM: Welcome to the natural language game creation program for general game playing! What size would you like your board to be?

USER: 3 by 3

SYSTEM: How many players does your game have?

USER: 2

SYSTEM: What types of pieces does player1 have?

USER: X

SYSTEM: Do any of player1's pieces start on the board?

USER: No

SYSTEM: What types of pieces does player2 have?

USER: O

SYSTEM: Do any of player2's pieces start on the board?

USER: No

SYSTEM: What can a player do on their turn?

USER: A player can place a piece in any open square

SYSTEM: How does a player win?

USER: A player wins if they get 3 pieces in-a-row

SYSTEM: Aside from a player winning, how does the game end?

USER: The game ends when the board is full

Many of the system's queries only require simple answers from the user, however it can be seen in the last three queries that the system is able to accept more complex natural language inputs.

4.2 Processing Input

The processing of input takes full advantage of the fact that dialog is being used, and the processing of input from each state of the dialog is tailored to the needs of that state. For example, when querying the user for the size of the game board or the number of players, using regular expressions is sufficient to determine the user's response since the number of possible answers is fairly limited. On the

other hand, when processing the answer to a question such as "How does a player win the game?", more complex natural language processing tools are needed. In this case, semantic grammars are used to parse the user input and then the resulting parse trees are analyzed in a way that allows the system to understand what the user meant and generate the correct GDL code.

4.2.1 Parsing

Currently, two custom designed semantic grammars are used for processing the more complex inputs. The first is used to parse inputs relating to how the players win and how the game ends, and the second is used to parse inputs about the moves a player may make on their turn. These were both designed by hand using descriptions of games taken from the General Game Playing website and iteratively building up grammar rules to cover the descriptions [17]. One of these grammars can be seen in Appendix B.

Continuing the Tic-Tac-Toe example from Section 4.1, parsing the sentence "A player wins if they get 3 pieces in-a-row" using the first semantic grammar will produce the following result:

```
(S
  (RESULT (ENTITY (PLAYER player)) (ACTION win))
  (IF if)
  (COND
    (ENTITY (PLAYER they))
    (ACTION
      (POSSESS get)
      (STATE (ENTITY (NUM 3) (PIECE piece)) (STATE in-a-row)))))
```

How this result is analyzed will be discussed further in the next section.

The pipeline for performing the parse is as follows: First, the input text is lowercased and then lemmatized using the NLTK WordNet lemmatization tool [18]. Next, stopwords are removed from the text and the remaining text is parsed using the NLTK ChartParser [18] and one of the custom grammars. Finally the NLTK parse trees are translated into a custom tree structure for further analysis.

4.2.2 Analyzing Parsed Input

The analysis of the resulting parse trees is done in three parts. In the first part, the tree is split up into important chunks that can each be analyzed individually. Continuing the previous example, the input "A player wins if they get 3 pieces in-a-row" has two main chunks — the RESULT sub-tree and the COND sub-tree — so these will be processed separately. Similarly if the conditional had multiple conditions then each condition would be processed separately.

The second part of analysis is looking for game concepts that the system knows inside of the different tree chunks. The system has a static list of these concepts, and currently this is done by simple keyword search. One such concept in the Tic-Tac-Toe example would be "in-a-row". Each of the concepts has certain slots that need to be filled — in the case of "in-a-row" there are slots for number, player, and piece — and the final step of analysis is to find the correct values to fill these slots.

Since these values may not be found within the current chunk of the tree, the search for these values is not limited to the current tree chunk, however values that are closer in the parse tree are given preference over those that are farther away. Furthermore, the slots may have default values in the case that the desired value does not appear anywhere in the parse tree. In the case of "A player wins if they get 3 pieces in-a-row" case the building block for "in-a-row" would be looking for a specific type of piece. However, since type of piece is not specified, it is assumed that any type of piece will satisfy the condition, and the piece type is left as a variable in the GDL code.

4.3 Intermediate Representation

The intermediate representation of the game is split into two parts. The first is a list containing functions that write blocks of GDL code and their arguments. This list is generated as the different dialog states are visited, with each state adding the functions relevant to that state and the user input.

The second part is a set of classes that store information about the different game components shared by all the games in the target domain. Specifically, all the games will have a board, player(s), and some form of piece(s). These classes are

used when outputting the GDL code to fill in pieces of information such as the size of the board and the number of players.

4.4 Outputting the GDL code

Once the intermediate representation of the game has been created, outputting the code is a fairly simple process. Since the list of functions that write blocks of GDL code has already been generated, this part of the system only has three tasks to perform. First, it has to call the functions in the list with the arguments provided by the intermediate representation. Some of the GDL code written by these functions is dependent on helper code written by other functions. Therefore, the second task this component of system must perform is to keep track of these dependencies and add other functions to the function list when needed. Lastly, the system keeps track of which functions have already been called to make sure the same GDL code isn't written multiple times. This isn't completely necessary to the functionality of the GDL code, but it keeps it cleaner and more readable for later inspection. A complete game description for the game generated by the Tic-Tac-Toe dialog in Section 4.1 can be seen in Appendix A.

The GDL code building blocks were written in a similar way to the semantic grammars. Many sample games in the GDL were examined to find general trends and then code was written to reflect these trends. In many cases it was possible to write code that would work for any game in the domain. For example, since well-formed games require that every player make a move at every step, there is usually a "noop" function written in games with multiple players so that players may do nothing when it is not their turn. Since this function is not game dependent, it can be written to work for any game as seen below:

```
(<= (legal ?player noop)
    (role ?player)
    (not (true (control ?player))))
```

Another example is the concept of adjacency. Since all games in the project

domain must be played on a rectangular grid of squares, whether two objects are adjacent will be the same for all games.

4.5 Verification of the GDL code

It is not enough to simply produce code that looks like GDL, so some form of verification is needed to show that the code being produced is actually usable GDL. This verification was done using the validator tools provided in the General Game Playing Base Package [19]. Figure 4.5 shows that code generated from the Tic-Tac-Toe dialog in Section 4.1 (seen in Appendix A) successfully passed all of the validation tests.

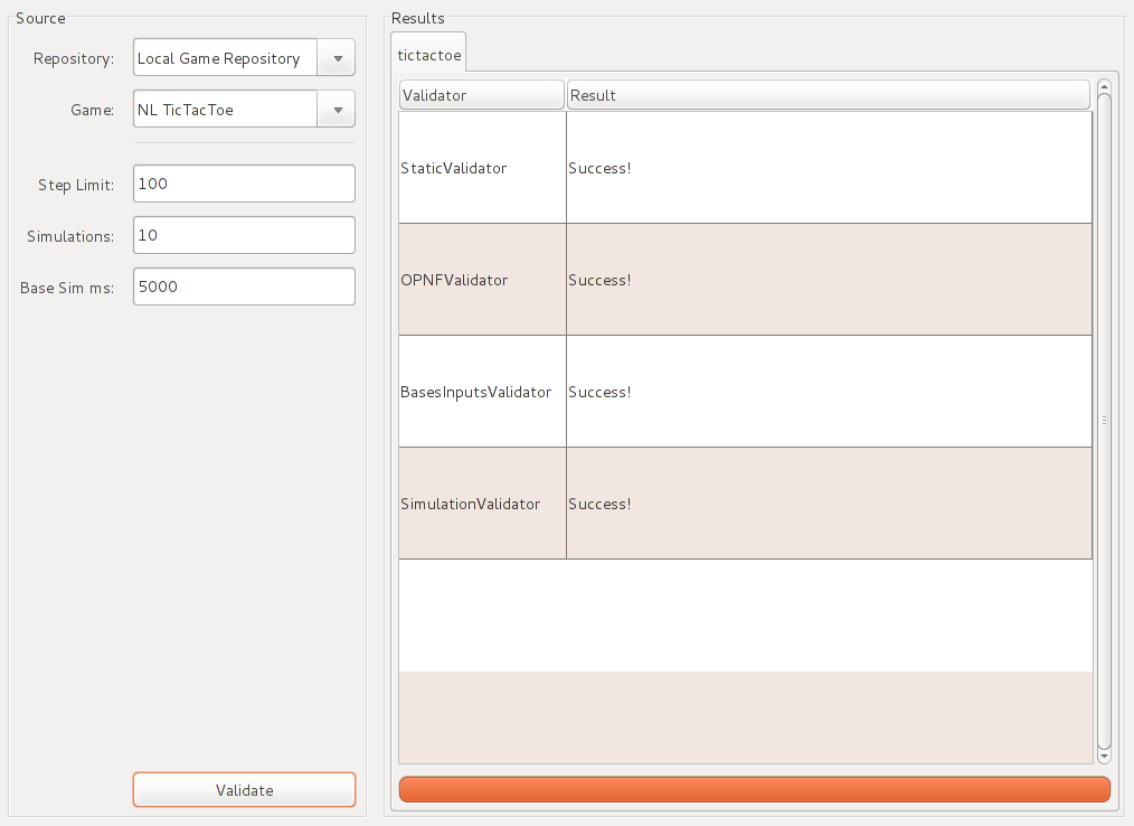

Figure 4.2: Successful Validation of Generated GDL for Tic-Tac-Toe

One thing to note is that this validation means that the code written is valid GDL code, but it does not mean that it accurately reflects the user's intention. This type of validation could only be done by having the user play the game until

they are satisfied that the game is correct. Unfortunately, this means that for even moderately complex games, errors in the game description may go unnoticed for long periods of time.

5. FUTURE WORK

In its current state, this project is just a proof of concept and is far from a complete system. As such, there are many things that could be improved. Some of these changes are less significant and would allow the system to be used more easily in the general game playing community. Other changes which are needed to truly open up the system to the general public, however, will require significant time and effort.

The easiest improvement to the system would be to add more building blocks for the user to build their game with. For example, adding more types of pieces or more conditions for winning or moving would greatly increase the domain of games the system is able to handle. At the same time, adding more building blocks would not require major changes to the current system's structure. Most of the work would be in writing the GDL code for these building blocks inside of functions that could then be called by the system when needed.

Another ongoing task that does not require major changes to the current system is improving the semantic grammars used for parsing. The coverage of the grammars is not currently the limiting factor in what inputs the system is able to handle. For example, the current semantic grammars can parse a more complex sentence like "The first player to get 3 pieces in-a-row in the center 3 x 3 square wins" however the later parts of the system would have no idea what "the center 3 x 3 square" means or even what in means to be "in" something. Even so, the coverage of the semantic grammars is something that will eventually need to be improved. In addition to improving the coverage of the grammars, the more ambiguity can be minimized, the easier it will be for later stages of the system.

Another improvement would be to make the grammars dynamic instead of static. This would allow the user to create new names for pieces and then be able to refer to those names later in the game creation process. If in addition to this change a new grammar was created for describing how pieces move on the board, the system would be able to at least parse the input for custom pieces — an important step in expanding the domain of games the system is able to create.

While these modifications would greatly improve the system and make it easier to use within the general game playing community, there are fundamental limitations with the system's current implementation. These issues require more radical changes if they are to be overcome and the system is to allow people with no programming experience to create games. The most obvious limitation is the fact that the possible games are ultimately limited to the building blocks available. Given enough time, it is foreseeable that a large portion of the games a user might desire to play would be covered by the system. However, to truly allow a user to describe any game the GDL is capable of representing — or even any game in the target domain — requires that the user is able to define new building blocks however they desire. This is an ambitious task with no obvious solution, and is therefore left open for now.

Another area where major changes are required is in the dialog with the user. The dialog as it is now forces the user to follow one predefined conversation and this approach is neither natural nor flexible. Ideally, the user should be able to describe the game in the way they want — at least to a certain degree. A more sophisticated finite state machine could help this, but moving to a frame-based approach and a more free-form dialog would be even better. A more free-form dialog might also allow the user to change what they said earlier in the conversation rather than having to start over the conversation every time they make a mistake or change their mind. Another problem with the dialog is that the system can handle neither anaphoric nor elliptical sentences. Adding support for these kinds of sentences is another step towards making conversation with the system feel more natural and less tedious.

A final possible improvement to the system would be to allow the user to refer games they have already created when creating new games. For example to create a game of Tic-Tac-Toe with three players the user could just say, "I want to play Tic-Tac-Toe with three players", instead of redefining the game from scratch. This could be done by storing the intermediate representation of each game, changing the relevant values for the new game, and then rewriting the code from the intermediate representation. Unfortunately, it would only be possible to refer to games that had already been created with this system, as it would be difficult to decipher arbitrary

GDL code in order to figure out which part of the code did what and then change the code appropriately.

5.1 Back to Blokus

In Section 1 an example was given of a user describing the game of Blokus. This section will examine some of the changes that would be required of this system — in terms of the future work described above — in order to allow such an interaction.

In broad terms, Blokus is a game that revolves around trying to place pieces of various shapes and sizes on a board until there is no legal way for any of the players to place another piece on the board. — more specifics can be found online[2]. One thing to note is that some the pieces in Blokus would be fairly difficult to describe, so these changes will assume that the system will know the pieces used in Blokus already (or at least be familiar with dominos, trominoes, tetrominos etc...) when the user is trying to describe the game.

The first change that would have to be implemented would be to create building blocks for the pieces in the GDL. For example, in order to know if a piece can be placed in a certain position, there has to be a description of what it would mean for there to be an open space in the shape of that piece. Below is how this code might look for the 2x2 square:

```
(<= (open 2x2_square ?col ?row)
    (succ ?col ?col2)
    (succ ?row ?row2)
    (open ?col ?row)
    (open ?col ?row2)
    (open ?col2 ?row)
    (open ?col2 ?row2))
```

In this case **succ** is a successor relation and (?col, ?row) would be the bottom left corner of the square. This same process would have to be repeated for each distinct orientation of all of the Blokus pieces.

[2]http://mattelgames.com/en-us/blokus/index.html (Retrieved July, 10, 2014)

Once these building blocks are in place, a few additions would need to be made to the semantic grammars. While the current grammar would be able to cover a sentence like "The game ends if no player can move", other input required to describe Blokus like "The player with the least squares at the end of the game wins" is not covered by the current grammars. Incorporating these new sentences into the grammar would be done in the same way that the grammar was originally built — through an iterative process of looking at sentences for trends and building rules to reflect these trends.

The final major change would be adding additional concepts to the system for use in analyzing the parsed input. For example, the system currently does not have any concept of what "least" means. Therefore, it would have to be added to the list of concepts with the appropiate slots — for example a slot for what player has the least and a slot for what object they have the least of — and a reference to a function in the GDL that would do comparisons of objects.

After these changes, the system would then be able to parse the input about Blokus, fill in the slots for the appropriate concepts, and produce the GDL associated with those concepts and the fill-in slots.

6. CONCLUSION

Overall, the system presented is a successful proof of concept for a system that can translate natural language into the Game Description Language. This is the first step that has been taken towards a system that will be able to take natural language descriptions of games and allow them to be played by a general game playing system. The scope of the system may be limited, but it is able to use dialog to get input from the user, process this input, and produce valid and complete GDL code in a way similar to how a more complete system would. There are still several open issues — for example figuring out how to allow users to define new code building blocks — but at the same time there is a clear way forward for expanding and improving large parts of the current system so that it can first be used as a tool in the general game playing community and eventually used by non-programmers. This potential for growth and improvement holds promise for the future of this project, and the possibility that it will eventually reach its ultimate goals.

REFERENCES

[1] H. Lieberman and H. Liu, "Feasibility studies for programming in natural language," in *End User Development*. Dordrecht, Netherlands: Springer, 2006, pp. 459–473.

[2] M. Genesereth and M. Thielscher, "General game playing," *Synthesis Lectures on Artificial Intell. and Mach. Learning*, vol. 8, no. 2, pp. 1–31, Mar. 2014.

[3] M. Genesereth, N. Love, and B. Pell, "General game playing: Overview of the aaai competition," *AI Magazine*, vol. 26, no. 2, p. 62, 2005.

[4] S. Edelkamp and P. Kissmann, "Symbolic classification of general two-player games," in *KI 2008: Advances in Artificial Intelligence*. Dortmund, Germany: Springer, 2008, pp. 185–192.

[5] N. Love, T. Hinrichs, D. Haley, E. Schkufza, and M. Genesereth, "General game playing: Game description language specification," Stanford Logic Group Computer Science Department Stanford University, Tech. Rep. LG-2006-01, Mar. 2008.

[6] I. Androutsopoulos, G. D. Ritchie, and P. Thanisch, "Natural language interfaces to databases–an introduction," *Natural Language Eng.*, vol. 1, no. 1, pp. 29–81, Mar. 1995.

[7] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 2nd ed. Noida, India: Pearson, 2009.

[8] M. Halpern, "Foundations of the case for natural-language programming," *IEEE Spectr.*, vol. 4, no. 3, pp. 140–149, Aug. 2009.

[9] N. E. Fuchs, K. Kaljurand, and G. Schneider, "Attempto controlled english meets the challenges of knowledge representation, reasoning, interoperability and user interfaces." in *FLAIRS Conf.*, 2006, pp. 664–669.

[10] T. Kuhn, "A survey and classification of controlled natural languages," *Computational Linguistics*, vol. 40, no. 1, pp. 121–170, Mar. 2014.

[11] M. P. Barnett and W. M. Ruhsam, "Snap: an experiment in natural language programming," in *Proc. Spring Joint Comput. Conf.*, 1969, pp. 75–87.

[12] Y. Bassil and A. Barbar, "Myprolang-my programming language: A template-driven automatic natural programming language," in *Proc. World Congr. on Eng. and Comput. Sci.* WCECS, 2008.

[13] P. Yin, "Natural language programming based on knowledge," in *Int. Conf. Artificial Intell. and Computational Intell.*, 2010, pp. 69–73.

[14] H. Liu and H. Lieberman, "Programmatic semantics for natural language interfaces," in *ACM Conf. Comput. and Human Interaction*, 2005, pp. 1597–1600.

[15] D. Vadas and J. R. Curran, "Programming with unrestricted natural language," in *Proc. Australasian Language Technol. Workshop*, 2005, pp. 191–199.

[16] D. Klein and C. D. Manning, "Accurate unlexicalized parsing," in *Proc. 41st Annu. Meeting Assoc. for Computational Linguistics*, 2003, pp. 423–430.

[17] S. Schreiber. (2014, Jun.) General game playing. [Online]. Available: http://www.ggp.org (Retrieved June, 25, 2014).

[18] S. Bird, E. Klein, and E. Loper, *Natural language processing with Python*. Sebastopool: O'Reilly Media, Inc., 2009.

[19] S. Schreiber and A. Landau. (2014, Jun.) The general game playing base package. [Online]. Available: https://github.com/ggp-org/ggp-base (Retrieved June, 18, 2014).

APPENDIX A
Generated Code for Tic-Tac-Toe

Below is an example program in GDL generated by a natural language description of Tic-Tac-Toe. NOTE: Comments were added and the order in which the code appears in the file was changed after the code was generated for clarity.

```
;;;;BOARD SETUP;;;;

(init (cell 1 1 none none))
(init (cell 1 2 none none))
(init (cell 1 3 none none))
(init (cell 2 1 none none))
(init (cell 2 2 none none))
(init (cell 2 3 none none))
(init (cell 3 1 none none))
(init (cell 3 2 none none))
(init (cell 3 3 none none))

(next_column 1 2)
(next_column 2 3)

(is_column 1)
(is_column 2)
(is_column 3)

(next_row 1 2)
(next_row 2 3)

(is_row 1)
(is_row 2)
(is_row 3)

(board_part row)
(board_part column)
(board_part square)
```

```
(board_part board)

;;;;PLAYER SETUP;;;;

(role player1)
(role player2)

(init (control player1))

(opponent player1 player2)
(opponent player2 player1)

(owns player1 X)
(owns player2 O)

;;;;NEXT STATES;;;;

(<= (next (control player1))
        (true (control player2)))
(<= (next (control player2))
        (true (control player1)))

(<= (next (cell ?col ?row ?cell_player ?occupant))
    (true (cell ?col ?row ?cell_player ?occupant))
    (does ?move_player (place ?occupant ?dest_col ?dest_row))
    (distinct_cells ?col ?row ?dest_col ?dest_row))

(<= (next (cell ?col ?row ?player ?occupant))
    (does ?player (place ?occupant ?col ?row)))

;;;;LEGAL STATES;;;;

(<= (legal ?player noop)
    (role ?player)
    (not (true (control ?player))))

(<= (legal ?player (place ?occupant ?col ?row))
        (owns ?player ?occupant)
```

```
        (true (control ?player))
        (open square ?col ?row))

;;;;GOALS;;;;

(<= (goal ?player 100)
    (win ?player))

(<= (goal ?player 50)
    (not (win ?player))
    (opponent ?player ?opponent)
    (not (win ?opponent))
    game_end)

(<= (goal ?player 0)
    (opponent ?player ?opponent)
    (win ?opponent))

(<= (win ?player)
        (3_in_a_row ?player ?piece))

;;;;TERMINAL STATES;;;;

(<= terminal
    game_end)

(<= game_end
    (win ?player))

(<= game_end
        (full board ?col ?row))

;;;;HELPER FUNCTIONS;;;;

(<= (open column ?col ?row)
    (is_row ?row)
    (true (cell ?col ?any ?player none)))
```

```
(<= (open row ?col ?row)
    (is_column ?col)
    (true (cell ?any ?row ?player none)))

(<= (open square ?col ?row)
    (true (cell ?col ?row ?player none)))

(<= (open board ?col ?row)
    (is_column ?col)
    (is_row ?row)
    (true (cell ?any ?any2 ?player none)))

(<= (3_in_a_row ?player ?piece)
        (owns ?player ?piece)
        (succ ?col1 ?col2)
        (succ ?col2 ?col3)
        (true (cell ?col1 ?row ?player ?occupant1))
        (true (cell ?col2 ?row ?player ?occupant2))
        (true (cell ?col3 ?row ?player ?occupant3)))

(<= (3_in_a_row ?player ?piece)
        (owns ?player ?piece)
        (succ ?row1 ?row2)
        (succ ?row2 ?row3)
        (true (cell ?col ?row1 ?player ?occupant1))
        (true (cell ?col ?row2 ?player ?occupant2))
        (true (cell ?col ?row3 ?player ?occupant3)))

(<= (3_in_a_row ?player ?piece)
        (owns ?player ?piece)
        (succ ?col1 ?col2)
        (succ ?row1 ?row2)
        (succ ?col2 ?col3)
        (succ ?row2 ?row3)
        (true (cell ?col1 ?row1 ?player ?occupant1))
        (true (cell ?col2 ?row2 ?player ?occupant2))
        (true (cell ?col3 ?row3 ?player ?occupant3)))
```

```
(<= (3_in_a_row ?player ?piece)
        (owns ?player ?piece)
        (succ ?col1 ?col2)
        (succ ?row1 ?row2)
        (succ ?col2 ?col3)
        (succ ?row2 ?row3)
        (true (cell ?col1 ?row3 ?player ?occupant1))
        (true (cell ?col2 ?row2 ?player ?occupant2))
        (true (cell ?col3 ?row1 ?player ?occupant3)))

(<= (distinct_cells ?col1 ?row1 ?col2 ?row2)
    (true (cell ?col1 ?row1 ?any ?any2))
    (true (cell ?col2 ?row2 ?any ?any2))
    (or (distinct ?col1 ?col2)
        (distinct ?row1 ?row2)))

(<= (full ?part ?col ?row)
    (is_column ?col)
    (is_row ?row)
    (board_part ?part)
    (not (open ?part ?col ?row)))

(succ 0 1)
(succ 1 2)
(succ 2 3)
(succ 3 4)
(succ 4 5)
(succ 5 6)
(succ 6 7)
(succ 7 8)
(succ 8 9)
(succ 9 10)

;;;;BASE AND INPUTS;;;;

(<= (base (cell ?col ?row ?player ?occupant))
    (is_column ?col)
    (is_row ?row)
```

```
    (role ?player)
    (owns ?player ?occupant))

(<= (base (cell ?col ?row none none))
    (is_column ?col)
    (is_row ?row))

(base (control player1))
(base (control player2))

(<= (input ?player noop)
    (role ?player))

(<= (input ?player (place ?occupant ?col ?row))
    (role ?player)
    (is_column ?col)
    (is_row ?row)
    (owns ?player ?occupant))
```

APPENDIX B
Semantic Grammar for Goal and Terminal Conditions

```
S -> IF COND THEN RESULT | RESULT IF COND | PLAYER ACTION RESULT
RESULT -> ACTION | ENTITY ACTION
COND -> COND AND COND | COND OR COND | CONDLIST AND COND | CONDLIST OR COND | STATE
     | NOT STATE | ENTITY ACTION
CONDLIST -> COND , CONDLIST | COND , COND
STATE -> ENTITY STATE | STATE POSITION_RELATION ENTITY | STATE TEMPORAL_RELATION
    STATE | NUM_COMP | ENTITY
ACTION -> NOT ACTION | POSSESS STATE | to ACTION | have ACTION | ACTION ENTITY |
    ACTION POSITION_RELATION ENTITY | ACTION TEMPORAL_RELATION STATE
ENTITY -> PLAYER | NUM PLAYER | PIECE | NUM PIECE | BOARD_PART | NUM BOARD_PART |
    TEMPORAL | NUM TEMPORAL | ENTITY PART_RELATION ENTITY | POSSESSION | NUM
    POSSESSION | ENTITY OR ENTITY | ENTITY AND ENTITY | ENTITYLIST OR ENTITY |
    ENTITYLIST AND ENTITY
ENTITYLIST -> ENTITY , ENTITYLIST | ENTITY , ENTITY
PIECE -> MOD PIECE
BOARD_PART -> MOD BOARD_PART
POSSESSION -> PLAYER ENTITY
NUM_COMP -> between NUM AND NUM ENTITY | NUM_COMP than NUM ENTITY | NUM_COMP ENTITY
     than ENTITY
IF -> if | when | after | by
THEN -> then | ,
NOT -> not
AND -> and
OR -> or
STATE -> in-a-row | in-order | full | occupied | open | EMPTY
ACTION -> win | lose | end | move | capture | place | mark | reach | drop
POSSESS -> get | have
BOARD_PART -> board | cell | square | row | column | side | diagonal
TEMPORAL -> turn | game | match | time | end | beginning
PLAYER -> player | player NUM | MOD player | they | their | opponent | whoever
PIECE -> pawn | rook | knight | bishop | queen | king | x | o | disc | piece | it
MOD -> most | least | first | last | middle | center | top | bottom | left | right
    | NUM x NUM | NUM by NUM | opposite | full | occupied | different | open |
    EMPTY
NUM_COMP -> more | less | greater | fewer
POSITION_RELATION -> on | in | to | into | onto
PART_RELATION -> of
TEMPORAL_RELATION -> at | when
NUM -> 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | one | two | three | four | five |
    six | seven | eight | nine | ten | no | neither | all | every | both
EMPTY -> empty | blank
```