SHRDLU: A Game Prototype Inspired by Winograd's Natural Language Understanding Work (Extended Version)

Santiago Ontañón
Drexel University
Philadelphia, Pennsylvania 19104
so367@drexel.edu

Abstract

This paper describes a game prototype called "SHRDLU" that explores the concept of designing a game around the ideas behind Winograd's original SHRDLU system. We briefly describe the main gameplay, as well as the natural language and inference architecture used by game NPCs.

Introduction

This paper describes a game prototype that explores the concept of designing a game around the ideas behind Winograd's original SHRDLU system. Specifically, we present an early prototype of a game called "SHRDLU", an adventure game where the player controls a character in a sci-fi setting world, and can talk to the game NPCs in plain natural language. The name of the game comes from the fact that NPCs behave in a similar manner to Winograd's SHRDLU (Winograd 1972) (in the rest of this paper, we will use SHRDLU to refer to our game and "Winograd's SHRDLU" to refer to the original AI system). These NPCs understand and generate natural language, and have reasoning capabilities beyond what is usual in standard NPCs in this type of games, thanks to a full first-order logic resolution engine.

As the game was inspired by Winograd's SHRDLU, all the current AI subsystems use symbolic AI approaches. However, the game is designed in a modular way, so that, for example, the natural language processing modules could, in principle, be replaced by a more modern approach in the future.

The remainder of this paper briefly describes SHRDLU's gameplay and the main AI architecture that brings SHRDLU's NPCs to life.

SHRDLU

SHRDLU[1] is a sci-fi adventure game with a gameplay that borrows ideas from the early *Sierra* graphic adventure games of the late 80s such as "Space Quest" (Crowe and Murphy 1986). The player finds itself in a space station in an unknown planet without remembering how did she get there. By interacting with three robot characters (Etaoin, Qwerty and Shrdlu), the player will unravel the mystery about her identity and about Aurora Station[2].

Figure 1: A screenshot of SHRDLU showing the main character talking with an NPC robot named Qwerty.

As shown in Figure 1 the key difference between SHRDLU and Sierra adventure games is that, while in Sierra adventures the player had to type commands to make the main character perform actions (like in earlier Zork-style text-based adventures), in SHRDLU typing is used to talk to other NPCs in a similar way as in Façade game (Mateas and Stern 2003), or MKULTRA (Horswill 2014).

Different from Façade, where the focus was on balancing user and author intent to maintain an emotionally intense, dramatic action, SHRDLU employs deep natural language parsing and generation. A more related game is Ian Horswill's MKULTRA, also based around natural language processing inspired by Winograd's SHRDLU, but with different game mechanics (such as belief injection, to solve puzzles by making NPCs perform the actions the players wants).

Moreover, several other computer games have used NLP

[1]SHRDLU can be found here `https://github.com/santiontanon/SHRDLU`

[2]A video demonstration of an early prototype of SHRDLU (version 1.1) can be found at `https://youtu.be/8FNBTs2yv4s`. At the moment of writing this paper, the latest version is version 1.10.

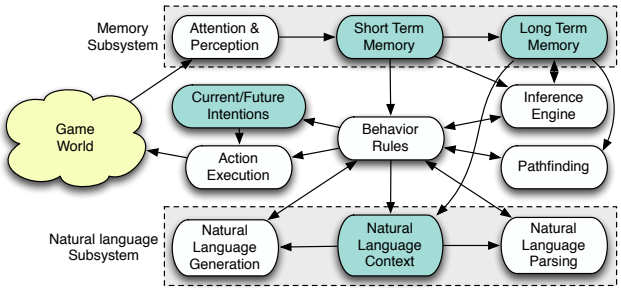

Figure 2: Main components of the AI architecture for the NPCs in SHRDLU.

techniques. For example *Event[0]* is a science fiction exploration video game where one of the core game mechanics is talking to a chatbot. A key difference between *Event[0]* and SHRDLU is that *Event[0]* employs chatbot-style techniques (basically keyword matching) in order to generate responses. SHRDLU on the other hand attempts to translate the sentence the player typed into an exact logical representation of it. In that way, you can ask the NPCs in SHRDLU to do things, or to give you information by formulating queries in natural language. On the flip side, while *Event[0]*'s techniques allows the game to let the players virtually type anything they wait, giving the appearance of understanding, SHRDLU's attempt to understand every word of what the player says, makes the types of sentences that can be understood necessarily more limited.

All of the text produced by the game NPCs is automatically generated from their internal logical representations (with an exception of a small easter egg, which is hard-coded), and thus, they do have a logical representation of what they are saying which they can reason about. Moreover, while scaling up NLP algorithms to handle arbitrary conversations is beyond the state of the art, the hypothesis behind the game's design is that it will provide an environment with is much richer than the original blocks world of Winograd's SHRDLU, but still manageable for current NLP techniques. Moreover, the NPCs have been kept to be robots purposefully to avoid the need for simulating emotions, and to provide a credible scenario where natural language parse errors are not to be unexpected.

The AI Behind the NPCs in SHRDLU

In addition to the player character, SHRDLU features three NPCs. Each NPC is controlled by a separate Winograd's SHRDLU-style AI system. Two of them are robots (Qwerty and SHrdlu) and one of them is a disembodied AI (Etaoin). Both types of NPCs use the same AI system, with the only difference being the set of actions that each is able to perform (robots can move around, pick up and drop items, etc., while Etaoin has control over some of the environmental systems such as lights, doors, etc.). Figure 2 shows the main modules of this AI system. The next subsections describe each of the modules in turn.

Memory Subsystem

SHRDLU uses a typed first-order logic knowledge representation to represent facts (e.g. "a table is a piece of furniture") and inference rules (e.g., "if an object is in a location X, and X is in Y, the object is also in Y"). This representation was designed specifically for the game and extends standard first-order logic by assuming the existence of an ontology O where each possible functor or constant is organized in a multiple-inheritance hierarchy. Every single object in the game (furniture, items, characters, locations, etc.), action that characters can perform, and properties of objects (colors, size, etc.) and relations ("on top of", "before", "owns", etc.) is represented in the ontology, so that the game state can be closely represented in memory.

Every NPC in the game has a selective attention procedure to prevent them from observing the whole game state (which makes it both more realistic, and also keeps the knowledge base manageable). All clauses representing perception are stored in a *short term memory* buffer, where they stay for only a fixed amount of time before they are forgotten. Certain perceptions (such as verbal utterances) are stored in long-term memory, which also contains background knowledge, and all the facts learned from dialog with other NPCs or with the player (the AI does not distinguish when they are talking to the player or to another NPC).

Logic Representation As mentioned before, SHRDLU uses a typed first-order logic knowledge representation. Thus, every variable or constant is associated with a type, or sort, s. The ontology of types is represented as a partially ordered set $O = \langle S, \preceq \rangle$, where S is a finite set of sorts, and $\preceq$ is a *more general than* relation. The special sort $\perp \in S$ (or *any*) is the most general sort, thus $\forall_{s \in S} \perp \preceq s$. Since O is defined as a partially ordered set, it has a semi-lattice structure, and thus supports multiple inheritance.

Let V be a potentially infinite set of variables, and C a potentially infinite set of constants (such as strings or numbers). A *term* t is of the form $t = f(s_1 : p_1, ..., s_n : p_n)$, where $f \in S$ is the *functor*, and p_1, ..., p_n are the arguments. Each argument p_i has a sort $s_i \in S$, and p_i is either a variable in V, a constant in C or another term.

A *sentence* s is an expression that can take any of the following forms:

- A term t.
- A negated sentence $\neg s$
- A conjunction: $s_1 \wedge s_2$, where s_1 and s_2 are sentences
- A disjunction: $s_1 \vee s_2$, where s_1 and s_2 are sentences

We do not use implication symbols, since those can be made out of disjunctions and negations. Additionally, all sentences are directly represented in CNF (conjunctive normal form), and thus no quantifiers are used (the conjunction connective was left in the language for convenience, and is converted to disjunctions before using it for inference). For example, the following sentence, part of the background knowledge in the NPCs of SHRDLU represents that an object can only be held by one character at the same time:

$\neg verb.have(\#id : x_1, \#id : x_3) \vee \neg verb.have(\#id : x_2, x_3) \vee = (x_1, x_2)$

Where $\#id$ is a sort specifying that the parameter represents the identifier of an object in the world $verb.have$ is the sort used to represent the verb *to have*, $=$ is the sort used to represent equality, and x_1, x_2, and x_3 are variables.

For convenience, $\neg$, $\vee$ and $\wedge$ are also sorts in the ontology, so that sentences can actually be represented as terms if necessary.

The ontology of SHRDLU contains sorts for all of the object types, properties, verbs and relations required to represent the game world. Additionally, some sorts, such as $\#id$, are used to denote game specific things, such as object identifiers.

Inference In order to perform inference, NPCs in SHRDLU are equipped with a standard resolution engine (Robinson 1965), which is in principle sound and complete for the typed first-order logic used in the game (although in practice it is not, since we added a collection of pruning rules to make inference efficient, as described below). For example, if the player asks "where is Qwerty?" to some NPC, a resolution process is started trying to see if there is any value of X for which the clause $space.at(\#id : qwerty, \#id : X)$ is entailed by the knowledge base.

The only difference with respect to a standard first-order-logic resolution engine is the ability to handle sorts. Thus, the resolution rule used is the following:

$$\frac{t \vee t_1^1 \vee ... \vee t_n^1 \quad \wedge \quad \neg t' \vee t_1^2 \vee ... \vee t_m^2 \quad \wedge \quad t\ \phi \equiv' t'}{(t_1^1 \vee ... \vee t_n^1 \vee t_2^1 \vee ... \vee t_n^2)\ \phi}$$

Where ϕ is a variable substitution through which the arguments t are made equivalent to the arguments of t', and the functor of t is either more general or more specific than that of t' (notice that since the ontology is a partial order, it is possible that none of these two conditions are satisfied, and in such case, we say that the functors do *not match*). The notation $\equiv'$ is used to represent this notion of equivalence (equivalent arguments, and functors that *match*, but are not necessarily identical. The notation $t\ \phi$ means the result of applying ϕ to t. Where a variable substitution is a mapping of variables to either other variables, constants or terms: $\phi = \{x_1 \rightarrow p_1, ..., x_n \rightarrow p_n\}$, where p_i is another variable, a constant or a term. Finally, terms that have the sort $=$ as the functor are handled in a special way, using a simplified version of the common *demodulation* strategy that is common in resolution engines (in this simplified version, equality terms are only handled once both their arguments have been resolved to constants).

Moreover, for practical reasons, the resolution engine used in the game uses two additional mechanisms to make inferences finish in a reasonable amount of time:

- Heuristic search pruning: Specifically, the game does not allow resolution to produce sentences with more than 6 terms, and also does not allow the production of sentences where the variable substitution ϕ only maps variables to other variables, nor the production of sentences that are larger than either of the two initial sentences. Additionally, inference processes that go beyond one million resolution application steps are interrupted. This ensures that most of the inferences that result from typical interactions can be solved, and that they finish in less than a few seconds, and usually instantaneously. The downside is that these heuristics make the inference process incomplete (there might be sentences that can be proven false or true, but that cannot be proven due to these pruning rules).
- Spatial predicates handling: whenever a sentence that contains spatial predicates representing concepts such as *north-of*, *inside-of*, *in-front-of*, *behind*, if the arguments of these terms are already resolved to constants, and if the time context at which we are applying resolution is the present time, then a hardcoded routine directly checks if the spatial relations are satisfied in the game environment and updates the sentences directly. This accelerates spatial inferences significantly in the cases where the hardcoded routine can be used. However, if this routine cannot determine the truth value of the term (because the object being referred to are outside of those modeled by the game engine, or because we are talking about the past or the future), then their truth evaluation is left to the standard resolution engine. Moreover, since spatial relation predicates are not included in the knowledge base (except for the *space-at* predicate), usually these queries are resolved unsatisfactorily. This is, however, another trade-off that had to be taken in order to keep the knowledge base size under control. Since the number of terms required to represent spatial relations grows at a quadratic rate given the number of objects in the game world.

Attention and Perception At every game step, the NPCs perceive the state of the objects in the game. This perception is represented as a collection of terms, that are stored in *short term memory*. Terms in short term memory stay there for a predefined short amount of time (currently set to 2 seconds), and then are forgotten. Moreover, given the very large number of objects present in the game (characters, rooms, furniture, items, etc.), if the current state of all those objects is stored in memory at the same time, inference processes become unmanageable., even with the pruning rules described above. Thus, an *attention* procedure was implemented so that NPCs only perceive a subset of the objects in the game at a given time.

For NPCs that have a physical body (robots), attention is implemented in a natural way: they can only see what is within a certain radius of them. For the disembodied Etaoin character, the way this is implemented is having a set of *focus* objects that Etaoin cares about (the player, the robots and a special item called the *communicator*, which is a wrist band that allows the player to talk to Etaoin even when outside of the building where Etaoin resides. Etaoin's attention is implemented by iterating over the set of focus items, and only perceiving what is within a certain radius of those objects. Moreover, the only object that can be perceived when outside of Etaoin's building is the communicator. If the player leaves that communicator inside, Etaoin will not be able to focus on the player when outside.

All the information concerning the game state is represented using the following types of terms:

- Object existence: when an NPC perceives an object, a term of the form $s_o(\#id : c_o)$ is added to short-term

memory, where s_o is the sort of the observed object (e.g., a *human*, a *table*, etc.), and c_o is a constant with a unique identifier for that object.

- Properties: objects have properties, for example an item might be *broken*. Properties are represented by sorts, and each perceived property s_p of an object with identifier c_o is represented by a term $s_p(\#id : c_o)$.
- Properties with value: some properties have value, for example, a chair has a color that can be grey, blue, etc. Properties with value are represented as: $s_p(\#id : c_o, s_v : p_v)$, where s_v is the sort of the value (e.g., *number*) and p_v is the value, which is often a constant.
- Relations: relations, such being in a room involve two objects. For example the term $space.at(\#id : 45, \#id : 834)$ states that object 45 is in object 834. For example, if 45 is a robot, and 834 a room, that means the robot is in the room.
- Relations with value: finally, some relations have values. For example, the *distance* between two objects is a relation with a value, and $distance(\#id : 45, \#id : 86, meter : 56)$ represents that objects 45 and 86 are 56 meters apart.

Handling Time: Although inferences concerning time are not fully handled by the inference engine of SHRDLU, there are a few mechanisms included in the NPCs AI to handle time inferences to some extend. Specifically, there is a special sort in the ontology called $\#stateSort$. Any term that has a functor that inherits from $\#stateSort$ is considered to be something that can change over time (for example, $space.at$ inherits from $\#stateSort$). Thus, if one such term is observed that has the same first argument as an already existing term in memory, the previous term in memory is overwritten, and the previous one (together with the time tag of when it was added) is moved to a historic memory, used to retrieve facts about the past, if needed. In this way, the memory of the NPCs only contains the most up-to-date state of the game. Inference processes can refer to the current time instant, or to a specific time instant in the past. When an inference process refers to a specific time instant, the state of the long term memory is restored to that instant by pulling facts from the historic memory, and then regular inference is performed. This cannot handle inference processes involving interactions between different time instants, but can handle basic queries such as "was I in the infirmary yesterday?". Moreover, another limitation of this system is that it can only handle specific instants of time, and not intervals. For example, when answering "was I in the infirmary yesterday?" NPCs in SHRDLU would restore their long term memories to the point in time when "yesterday" started, and answer based on that. So, if I happened to have been in the infirmary at some other time yesterday, the answer will not be correct. Time inferences will be improved in future versions, but the current solution allows for efficient, even if limited, inferences.

Short and Long Term Memory NPCs have two memory storages: the *short term memory* (STM) and the *long term memory* (LTM). The short term memory consists of a set of *terms* and is used only to store everything that is perceived by the attention and perception module (terms in STM remain there for 2 seconds unless they are perceived again).

The long term memory consists of a set of *sentences*, and content stored there is kept forever. Every term that is perceived in the game is copied to short term memory, and from those, a subset is further moved to long term memory, as determined by what we call the STM2LTM filter. Specifically, the current version of the STM2LTM filter is a simple rule that just copied to long term memory all those terms representing sentences said by either the player or an NPC (the AI does not know that some of the other characters in the game are NPCs and one is controlled by the player; they are all perceived in the same way).

The long term memory is seeded when NPCs are created with a collection of background knowledge rules, such as objects cannot be in two locations at the same time, or the fact that if an object A is in B, and B is in C, then A must be also in C, etc. Additionally, terms representing knowledge that the NPCs must have (such as knowledge about the planet where the story occurs, etc.) is also loaded in at startup.

Behavior Rules

Each NPC has a set of *current intentions*, which are actions it is currently trying to execute (not all of these actions need to be physical actions, and could be, for example, launching inference processes), and a queue of *future intentions*, which will move, one by one, to the set of current intentions as soon soon as the current intentions set is empty.

An important type of the behavior of NPCs is communication with other NPCs/players. SHRDLU uses a formalism based on *speech act* theory (Searle 1969) to represent the different utterances that players or NPCs can say in natural language. NPCs in the current version of SHRDLU can recognize and generate 30 different speech acts that include things like: *greet*, *farewell*, *inform*, *acknowledge*, etc. (Table 1 for a complete list).

A collection of "behavior rules" dictate how the different internal memory structures of an NPC are updated upon receiving each different speech act, and if any action needs to be pushed to the set of current intentions or future intentions as a response. Some speech acts request the NPCs to perform actions some trigger inference processes to produce answers, some just add facts or rules to memory, and others just update the conversation state. Some of these actions call upon the different reasoning mechanisms the NPCs have (As shown in Figure 2), such as the resolution engine or the pathfinding module. For implementation reasons, however, the pathfinding module used by the NPC AIs is not the one actually used for moving around the map, since the game is implemented on top of a previous game engine we created for a different game which already had a pathfinding algorithm incorporated. Thus the NPCs' AI pathfinding module is only used for reasoning purposes (e.g., if the player asks how to get from one point to another), and uses the representation of the game map in long term memory, rather than that used by the game engine.

Speech Act	Example
callattention	Shrdlu!
greet	Hello Shrdlu!
farewell	See you later!
thankyou	Thanks!
youarewelcome	You are welcome!
inform	This chair is small
inform.parseerror	I do not understand fsfudgs
inform.answer	Yes
acknowledge.ok	Ok!
acknowledge.contradict	That cannot be true
acknowledge.invalidanswer	That does not answer my question
acknowledge.denyrequest	I cannot do that
sentiment.good	Good!
sentiment.bad	That's bad!
sentiment.surprise	Really?!
request.action	Please, open the door
question.howareyou	How are you doing?
question.predicate	Do you have the garage key?
question.whereis	Where is Shrdlu?
question.whereto	Where should I go to find a key??
question.query	What is your name?
question.whois.name	Who is Shrdlu?
question.whois.noname	Who is the red robot?
question.whatis.name	What is Aurora Station?
question.whatis.noname	What is this thing?
question.action	Can you open the door?
question.when	What time is it?
question.howmany	How many chairs are in the infirmary?
question.why	Why did Shrdlu leave?
question.how	How do I fix the spacesuit?
moreresults	What else?

Table 1: Speech acts recognized by the current prototype.

Natural Language Subsystem

This is the most complex component of SHRDLU's NPC AI architecture and is organized in three main modules: *natural language context*, *natural language parsing*, and *natural language generation*. All these three modules rely on representing the different utterances the player or the NPCs produce in a logical representation based on *speech acts* (Searle 1969). Thus, we will start by describing the speech act system of SHRDLU.

Speech Acts in SHRDLU. Every utterance in natural language perceived from another characters (player or NPCs) in SHRDLU is (attempted to be) translated to a *speech act*. Speech acts thus represent the different things that characters can say. There are many different types of speech acts such as: greeting, asking a question, stating an answer, etc. The literature of natural language processing has used several different names for these speech acts, such as dialogue acts or dialogue moves, among others.

Table 1 lists all the speech acts currently recognized by the natural language parser used in SHRDLU. Each speech act, moreover, has parameters. For example:

- $greet(C : [any])$: this is the prototype of the *greet* speech act, which has a single parameter (C), representing who are we greeting. For instance, the sentence "Hello!" is just represented as $greet(C : [any])$, but the sentence "Hello Shrdlu!" is represented as $greet(`shrdlu' : [\#id])$.
- $question.query(C : [any], X : [any], Q : [any])$: this is the prototype for questions of type "query", which requests an answer to a question that needs to find the value of a specific variable X which makes a logical formula Q satisfiable according to the knowledge base. For example, the question "what is your name?" (if asked to, say, Etaoin) would be represented as $question.query('etaoin'[\#id], X, name(`etaoin'[\#id], X))$.

Given the open-ended nature of the conversations a player might expect to have with the NPCs in SHRDLU, a large number of speech acts is required, as reflected in Table 1, when compared to the set of speech acts or dialogue moves used by other systems. For example, the GoDiS system (Larsson et al. 2000) employed only 8 types of dialogue moves: *ask*, *answer*, *repeat*, *request_repeat*, *greet*, *goodbye*, *thank* and *quit*. Notice, for example that the *repeat*, *request_repeat* and *quit* types are not present in SHRDLU, since those are more typical of task-oriented dialogue systems that serve the purpose of helping a user accomplishing a task such as booking a restaurant. Moreover, given the complexity of the physical world in SHRDLU, there are many different types of question speech acts. These are separated, since the behavior that is expected to produce an answer to these is very different. For example, predicate questions require testing if a logical statement is true or false, query questions require finding the value of a variable, when questions require finding the time at which a given event was added to the knowledge base, etc.

Moreover, notice that, in principle, it might seem that the query question should be the most general type and most others are just particular cases. However, we decided to add different types of questions, such as *question.whereis*, or *question.action*, since they vary significantly from answering a query. For example, if *question.whereis* was implemented as a *question.query*, the logical formula would have to capture common-sense things such that when we ask where is something, we want to know the most specific location (i.e., answering that something is "in the universe" would be a valid answer, but is not what the player is expecting). These can be encoded in the query, but would complicate inference. Thus, the different types of question encode all of these differences to strike a balance between generality and query answer efficiency.

Other types of questions would be hard to represent as queries. For example, upon a *question.how* type of question, the player would expect to receive a plan on how to achieve something.

The main task of the natural language parsing engine is thus, not to produce a parse tree as many other natural language parsers, but to translate a natural language sentence into a logical representation of a speech act. This means, that for example, a phrase "that red chair" will be translated to just the identifier of the corresponding real world object,

and the fact that the natural language text used the adjective "red" to distinguish the chair from the other chairs will not be captured in the performative.

Natural Language Context *Natural Language Context*: this is a record of the current state of a conversation. NPCs hold one *conversation context* per character they talk to. An assumption in the game is that conversations only happen between 2 characters at a time, conversations with three or more simultaneous characters are not supported. In the current version of the game, the context includes:

- Speaker: the ID of the character we are talking to.
- Conversation history: a time-tagged list of speech acts exchanged between the two characters.
- Mention history: a time-tagged list of entities (objects, locations, characters, etc.) mentioned in the conversation history, sorted by the time at which they were mentioned.
- Perception: this is updated upon receiving a speech act from another agent, and contains the list of entities in the perception buffer of the NPC at the time of receiving the speech act. This list is sorted by distance to the NPC.
- Question/Request stacks: each time the NPC asks a question or issues a request, it expects an answer. Questions/requests can be nested, and those that are still awaiting an answer are placed in a stack until answered (this stack is also used to repeat the question if enough time has gone by without an answer).
- Conversation state: this contains a list of flags to indicate whether we are in an active conversation with the other character, or if we had properly said good bye; whether we are expecting the other character to say thank you or you are welcome, etc.

The Mention history and Perception lists are used to disambiguate pronouns or other referring expressions during language parsing. For example, if the player says "give me the key", but there are two keys. If one of the keys appears earlier in the mention history list or is significantly closer to the NPC according to the Perception list, then those facts are used to disambiguate which key is being talked about. This is also used to disambiguate "this" vs "that", and other pronouns and articles.

The conversation history and the conversation state are used for managing the dialogue and generating appropriate expected responses such as "thank you". This is also used to determine if a character is speaking to us or not. In SHRDLU, the only thing that NPCs perceive with respect to communication is that another character said something in natural language (even when two NPCs talk to each other, their only means of communication is via natural language using their NLP and NLG modules). After parsing this natural language, they need to determine if the sentence was targeted to them. If the sentence contains an explicit mention to them, then it's easy, otherwise, the conversation state is used to determine that.

Natural Language Parsing The goal of this module is to translate a natural language sentence into one or a collection of speech acts. The module uses a grammar defined by a collection of parsing rules for each speech act (and collections of generic rules for noun phrases, verb phrases, etc. that are used by the other rules). The key differences between this custom parser with respect to standard NLP parsers such as the Stanford Parser (Klein and Manning 2003) are: 1) the output of parsing is not a parse tree of the sentence, but a logical representation of the speech act (e.g., "the red key" will get translated to the identifier "key1", and the fact that the natural language sentence used the adjective "red" is not captured in the output logical representation); 2) the parser employed is a "situated parser", as parsing rules include calls to special *de-reference* functions, which search for objects in the natural language context that match with referring expressions such as "the red key", or "this door". There are three types of de-reference functions (context dereference, hypothetical dereference and query dereference) that are used to translate phrases into logical descriptions. Thus, parsing and translation to logic is done in a single step as part of the parsing rules.

For example, two of the rules in the grammar are shown in Figure 3. We can see that each rule is defined by a pattern (to be matched by the natural language text) and a logical term, which will be the final output of parsing. Rules also have a numerical priority to disambiguate when there are multiple possible parses. For example, the first rule shows a pattern where first a *nounPhrase* will try to be found (this is a recursive call to see if there is any rules which produce a term with functor *nounPhrase* that match the current sentence. After that a conjugation of the verb *to be* is expected that matches in number (N), and person (P) with the noun phrase. After that either an adjective or a a phrase with an indefinite article followed by a noun are expected. If all of this matches, then C (which is a logical representation of the content of the noun phrase) is given to a special dereference function called *#derefFromContext* which tries to find any entity in the current conversation context, which matches the noun phrase. If this entity is found, then the corresponding logical term is returned as the output of parsing. For example, this rule will parse a sentence like "the chair is small", and return something like $perf.inform(LISTENER, small('45'[\#id]))$ (assuming the listener is the player, and the chair that matched was the one with id 45). Notice that the variable *LISTENER* was not resolved, and that's because the sentence does not specify who are we talking to. If the sentence was, instead "Shrdlu, the chair is small", then a different rule from the grammar would have matched with this sentence and replaced *LISTENER* by the ID corresponding to Shrdlu. Also, there is a special variable *SPEAKER*, which always resolves to the character that produced the sentence.

The second rule in in Figure 3 uses a different dereference function (*derefUniversal*), which, rather than finding specific entities in the natural language context, it checks whether the noun phrase can be represented by a universal expression, for example, the phrase "all chairs" will be represented by $chair(X)$. Thus, this rule will turn a sentence like "all chairs are small" to $perf.inform(LISTENER, \neg chair(X) \vee small(X))$, which

Pattern: $nounPhrase(X, N, P, C)\ \ verb('verb.be'[symbol], N, P, T)$
$(adjective(X2)\ |\ indefiniteArticle(ART, [singular])\ \ noun(X2, N))$
$\#derefFromContext(C, SUBJECT)$
Logical Term: $perf.inform(LISTENER, X2(SUBJECT))$

Pattern: $nounPhrase(X, N, P, C)\ \ verb('verb.be'[symbol], N, P, T)$
$(adjective(X2)\ |\ indefiniteArticle(A, N2 : [singular])\ \ noun(X2, N2))$
$\#derefUniversal(C, V, LEFTSIDE)$
Logical Term: $perf.inform(LISTENER, \neg LEFTSIDE \vee X2(V : [\#id])))$

Figure 3: Example rules from the natural language parsing grammar.

is the logical equivalent.

As of version 1.10, the grammar contains 287 rules. For efficiency during parsing, these rules are compiled to a finite state machine (FSM) in order to parse a sentence in a single pass, rather than having to try to apply rules one by one. Transitions in this FSM correspond to the different tokens in the rules, such as $noun(X2, N2))$, and terminal states contain the logical terms to be produced as output. The resulting FSM currently contains 3860 states.

Finally, before applying the parsing rules, the raw sentence is first processed through a tokenizer that identifies the part of speech of every word, and identifies if there are any multi-word tokens (e.g., "in front of" is a three word preposition, which should be considered as a single token).

Natural Language Generation This module translates speech acts represented in logical form to natural language. Although generation could probably be handled by reversing the parsing rules, text generation is currently performed by a collection of specialized routines. One such routine exists for each possible speech act.

The complete flow of events from the time an NPC perceives a sentence in natural language from the player or from another NPC up to the point when a response is produced (if any) is illustrated in Figure 4. As can be seen, the process is initiated when an NPC perceives a "talk" action by some other character in the game. At this point, the conversation context for the character that performed the "talk" action is updated with the latest information from short term memory and perception, and then natural language parsing starts. After tokenization, a dictionary-based multi-token detection routine identifies tokens that are made up of more than one word, and after that the sentence is annotated with part of speech tags. All possible part of speech tags for each token are annotated (to allow for maximum flexibility when applying the parsing rules). Also, if some sentence can be parsed in different ways depending on whether a sequence of tokens is considered a multi-word token or a sequence of individual word tokens, then all of those alternative taggings are also considered. The results is a list of possible part of speech taggings for the sentence (represented as a tree or alternatives). Once that is complete, the result is ran through the parsing rules described above, which make use of the different dereferencing functions, resulting in a parsed performative (or in a parsing error).

Once the NPC has the sentence parsed, it needs to identify if the sentence is directed to itself, and if it is, a collection of behavior rules dispatch the performative to the appropriate action handler. This handler could make use of any of the different reasoning modules, such as pathfinding, or inference. Once the action handler is done, its output is a collection of *intentions* that the NPC will try to execute as a response. These intentions could include performing any action on the physical world, or produce a response in natural language.

What Works and What Does not Work

After initial examination of what players do when playing SHRDLU, some conclusions can be drawn. Although most players responded positively to the game and to the form of interaction that the natural language capabilities of the game allow, there are still many aspects of the AI and game design that did not work as expected.

- Mis-interpretation of the interaction paradigm: several users interacted with the system as if it was an interactive fiction-style game, trying to issue commands such as "open door". Thus, the game design will need to be revised to emphasize that all text entered by the player results on the main player character "saying these words out lout", and that this text will not translate into actions to be performed by the player.
- Limits of natural language parsing: the current version of the parser has a rather limited vocabulary and grammar. As mentioned about, there are only 271 parsing rules in the current grammar (and the part of speech database, only contains 1721 entries at this point). This resulted in many sentences not properly parsed, creating frustration. This was specially true for players who played the earlier versions of the game, and later versions have minimized this problem. However, the game still expects somehow grammatically correct sentences, and players often type non-grammatical sentences since those are common in coloquial language. Other players thought the game was using a classic "keyword" based NLP and attempted non-grammatical sentences such as "where key", which were not recognizer either. The grammar has been expanded to recognize many of these (for example, most parsing rules do not expect matching number between verbs, articles and nouns any more). The latest versions of the game achieved a much higher success rate in parsing the sentences asked by the players. Moreover, players usually adapted, by trying to ask the same question in different

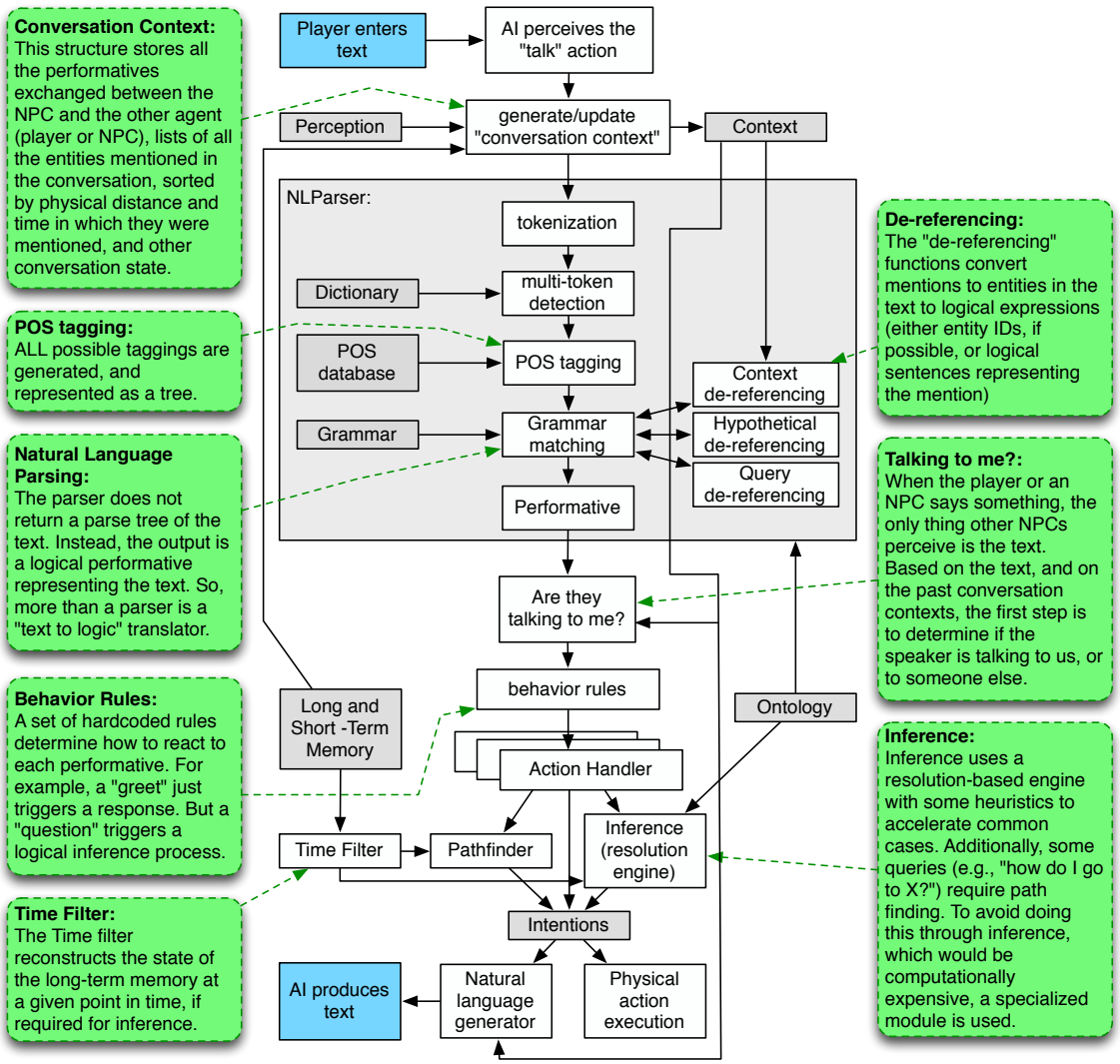

Figure 4: A flowchart of the different processes that take part in an NPC reacting to an utterance in natural language from the player.

ways until getting a proper answer.

- Reasoning limitations: the choice of representation for the game world was not sufficient for some of the queries that players might want to ask. In particular, time is not fully supported by the inference engine, and many time related queries do not result in satisfactory answers.

Conclusions and Future Work

This paper has presented SHRDLU, a game prototype exploring the type of gameplay that can be achieved by using Winograd's SHRDLU-style NPC AI. The current version of the prototype includes the whole first "act" of the game (out of a total of 4 for the complete storyline). Although no systematic nor thorough evaluation has been conducted at this point, initial evaluation with players close to us indicates that players are able to communicate with the NPC to get them to do what they need, and can get the answers necessary to progress in the story. We have also seen that the constrained story world of the game is, while much broader than the simple blocks world of Winograd's SHRDLU, still constrained enough for making natural language parsing feasible with a reasonable degree of success (although there is a lot of room for improvement).

As part of our future work, we plan to expand the set of speech acts that is currently recognized by the natural language subsystem. Once the prototype reaches a sufficient level of maturity, we would also like to open-source the engine to allow for other SHRDLU-like games by just configuring the environment, the knowledge bases of the NPCs, and the game script.

Acknowledgements: We would like to thank the people that helped beta test the current prototype (in chronological order in which they tested the game): Jichen Zhu, Josep Valls-Vargas, Jordi Sureda, Sam Snodgrass, Javier

Torres, Adam Summerville, Ahmed Khalifa, Sri Krishna, and Bayan Mashat.

References

Crowe, M., and Murphy, S. 1986. Space Quest: Chapter I The Sarien Encounter. *Sierra On-Line*.

Horswill, I. 2014. Game design for classical ai. In *Tenth Artificial Intelligence and Interactive Digital Entertainment Conference*.

Klein, D., and Manning, C. D. 2003. Accurate unlexicalized parsing. In *Proceedings of the 41st annual meeting of the association for computational linguistics*.

Larsson, S.; Ljunglöf, P.; Cooper, R.; Engdahl, E.; and Ericsson, S. 2000. Godis: an accommodating dialogue system. In *Proceedings of the 2000 ANLP/NAACL Workshop on Conversational systems-Volume 3*, 7–10. Association for Computational Linguistics.

Mateas, M., and Stern, A. 2003. Façade: An experiment in building a fully-realized interactive drama. In *Game developers conference*, volume 2, 4–8.

Robinson, J. A. 1965. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)* 12(1):23–41.

Searle, J. R. 1969. *Speech acts: An essay in the philosophy of language*, volume 626. Cambridge university press.

Winograd, T. 1972. Understanding natural language. *Cognitive psychology* 3(1):1–191.