Inquiry: The University of Arkansas Undergraduate Research Journal

Volume 7 Article 14

Fall 2006

Dynamic Composition of Agent Grammars

Kyle Neumeier
University of Arkansas, Fayetteville

DYNAMIC COMPOSITION OF AGENT GRAMMARS

By Kyle Neumeier
Department of Computer Science and Computer Engineering

Faculty Mentor: Craig Thompson
Department of Computer Science and Computer Engineering

Abstract:

In the very near future, as pervasive computing takes root, there will be an explosion of everyday objects that are uniquely identifiable and wrapped by a computational layer – effectively bringing the object to life. An important component of this system is the mechanism that will allow humans to interface with the objects. Menu Based Natural Language Interfaces (MBNLI) seem like a good candidate for this job because of the intuitive way in which they allow the user to build commands. However, the MBNLI system will have to scale with the number of objects in the system. This project describes context free grammar modules which are small grammar files that can be composed to form a larger grammar. Grammar modules can then be associated with individual objects, and in this way allow the MBNLI to scale according to the size of the system.

1 Introduction:

1.1 *Problem*

In a world where electronic devices are shrinking rapidly and the ability to associate unique identifications with objects is becoming commonplace, computing is escaping the desktop and becoming pervasive. The idea behind pervasive computing is that computers are migrating quickly from the familiar desktops, laptops, and PDAs that people use and are being incorporated into everyday objects. These objects – refrigerators, coffeemakers, sprinkler systems, web pages, etc. – will look the same as they do today, but they will be wrapped in a computational process that will give them the ability to communicate with people, the outside world, and even other objects via a programmatic interface and often a wireless connection [1].

In order to scale such a collection of network devices, or more generally a system of *agents* – everyday objects with a computational wrapper – humans will need some sort of control mechanism(s). One possible solution is to associate an interface grammar, or a set of rules describing the object's command set, with each object. The grammar could then be downloaded into a sort of next generation remote control which could in turn be used by a human to control any object that had such a grammar [2]. In order to accomplish this, the grammars would need to be able to be composed so that larger grammars could be synthesized from smaller ones in such a way that allowed for the ability of grammars to be "plugged in" and allowed for the reuse of common grammars.

Furthermore, the actual mechanism of communication with these agents must be intuitive. It is reasonable to think that an easy, familiar way for humans to communicate with machines is through the same mechanism that we use to communicate with each other, that is, through our natural languages (e.g. English). Therefore, the system of grammars that allow a user to control the agent should allow commands to be issued via natural language.

1.2 *Objective*

The objective of this project is to define a system for creating multiple grammars (called grammar modules) that can be composed to build a larger grammar. Such a system would provide a natural language interface for a multi-agent architecture.

1.3 *Scope*

The focus of this thesis is dynamic grammar composition for use with a menu-based natural language system (described below). A smart home RFID application is assumed but the results from this thesis are relevant to a much broader collection of applications that could include asset management, robots, semantic web, and other applications.

1.4 *Organization of this Thesis*

Chapter 2 covers background information about Menu Based Natural Language Interfaces (MBNLI), the Everything is Alive (EiA) architecture, and grammar composition. Chapter 3 explains an approach to creating a system of composable grammars via context free grammar modules. Chapter 4 discusses the implementation of the system. Chapter 5 states the conclusions and future work.

2 Background:

2.1 *Key Concepts*

In order to better explain the problem that a distributed grammar would solve and the method of creating a system of composable grammars, a brief overview of a related concept is provided. A discussion of the Everything is Alive project will clarify the reason that a composable grammar system is desired in the first place. Then an introduction to Menu Based Natural Language Interfaces will show how grammars can be used to build a natural language interface to an object. Next, context free grammars are reviewed and a few terms relating to composable grammars are defined. Finally, attributed grammars will be introduced.

2.1.1 *Everything is Alive Agent System Project*

The Everything is Alive (EiA) project at the University of Arkansas aims to develop a road map of pervasive computing. As an ever increasing number of objects become identifiable by a computer through technologies such as RFID and IPv6, there will be a tremendous explosion of things that we can control via a computer [1]. One goal of the EiA project is to develop an architecture that organizes these objects with a communication wrapper, which can be viewed as a kind of agent, into a system that optimizes usability and scalability [3].

An example of a system of agents at work in the context of the EiA world is a smart sprinkler system. Imagine that a flower bed agent has a sensor device that can measure the amount of water in its soil. When the soil is too dry for its flowers, it sends a message to the sprinkler agent that tells it to turn on. The sprinkler then asks the weather agent on the Internet if it will rain in the next 24 hours. If no rain is likely, then the sprinkler agent turns on and waters the flowers [1]. A second example is a thermostat that knows to turn the air conditioner on when a light turns on[1]. The point is that the objects are the same as before – the sprinkler is still a sprinkler and the thermostat is still a thermostat – but an agent wraps the object, effectively bringing it to life.

Past work in the EiA project has produced an architecture that facilitates agent-to-agent communication [4]. The agent class "wraps" an object and allows it to communicate with other agents by making available methods that send and receive XML messages. When XML messages are received by an agent, they are translated into the underlying object's native language so that the command can be executed. A few agents have been constructed, including an RFID reader agent that facilitates commands such as "turn on," "read for 300 ms" and "turn off [5]."

2.1.2 *Menu Based Natural Language Interfaces*

Building natural language interfaces to machines has been a grand challenge problem for almost as long as computers have existed. The idea is simple; it would be nice if a user could control a computer simply by speaking, as if to another person. The implementation, however, has proved to be extraordinarily difficult. Problems arise because computers cannot interpret connotations, clichés, body language, and idioms that all contribute to our understanding of language. The resulting situation is that the user either *overshoots* the ability of the NLI system by phrasing a command that cannot be understood, or *undershoots* the ability of the underlying system by not using features that are available because the user is not aware of them or not sure how to phrase the command to use them. This mismatch between the user's phrasing and the NLI system's capability is known as the habitability problem [6].

Menu Based Natural Language Interfaces (MBNLI) relieve the habitability problem by employing a predictive parser and cascading menus to present the user with a list of next possible choices. When the user selects a phrase from the menu, the parser generates a new list of next possible choices. This process continues until a complete sentence or command is built. Such a system guarantees that any command the user builds is syntactically correct. Furthermore, the user can develop an understanding of the capabilities of the underlying system by exploring the menus [6].

One particularly useful application of MBNLI technology is when it is used as the front end to a relational database. The interface enables users to build queries in English rather than SQL, allowing users who know nothing about SQL or about the database schema (i.e. relation and attribute names) to extract useful information from the database. In the past, NLIs to databases have typically been question and answer systems that allow users to ask natural language questions; however, these systems suffer from the habitability problem in that a user does not know what is able to be asked and exactly how to ask it [7].

The Everything is Alive project has found its own use for MBNLI. The project envisions a world in which many objects in the form of agents can be controlled by humans. Thus, an easy and standard way of issuing commands will be needed. Menu Based Natural Language Interface technology offers a solution to this problem. If all agents had an associated grammar, an MBNLI interface could be generated based on the grammar. The user would then be able to issue syntactically correct commands by building them. Furthermore, the user would know exactly what capabilities the agent has (by virtue of the cascading menus) [8].

2.1.3 *LingoLogic*

LingoLogic is an implementation of an MBNLI system created by Object Services and Consulting, Inc. It was based off of an earlier implementation developed at Texas Instruments in the 1980s called NLMenu. The intent of the project was to develop an MBNLI generator for a relational database. The idea was that if a static grammar and translation for SQL was developed, MBNLI interfaces could be generated relatively easily by providing the parser with a description of the particular database's schema [7].

LingoLogic consists of a front-end, implemented in Java, and a parser, implemented in C. The parser works like a LISP interpreter in that it is basically a read-eval-print loop. In other words, a command is given to the parser which processes the command immediately. The front end is a GUI that allows users to build commands and queries via a cascading menu. When a user has selected a word or phrase, the front end passes the selected item to the parser via a port, the parser predicts a set of next legal phrases, and sends them back to the front end. This process is repeated until a complete sentence is built. If a target language is specified, the parser can translate the complete sentence into the language. It is then the task of the front end to execute the translation (in whatever sense is appropriate). Because LingoLogic was intended as an interface generator for relational databases, its front end has the ability to execute SQL queries against a database and display the results [7].

2.1.4 Context Free Grammars

LingoLogic uses attributed context free grammars (CFG) to specify the syntax of commands and queries. Context free grammars consist of a finite set of terminals (T), a start symbol (S) which is a member of V, a finite set of nonterminals (V), and a finite set of production rules (P) that represent the recursive definition of a language. The productions take the following form:

Left Hand Side (LHS) -> Right Hand Side (RHS)

The LHS is a nonterminal that is being defined. The RHS is a string of terminals and non terminals that represent a way to form the LHS. Formally, the production rules may be defined as follows [9]:

$\alpha \rightarrow \beta$, where $\alpha \in V$, and $\beta = (T \cup V)^*$

A very simple version of the English language can be specified using a CFG[2].

S -> nounPhrase verbPhrase

nounPhrase -> article noun

verbPhrase -> verb nounPhrase

article -> THE | A

noun -> DOG | CAT

verb -> EATS | CHASES

Example 1: A simple English grammar

This CFG allows the construction of sentences such as "The dog chases a cat" and "The cat eats the dog."

2.1.5 Context Free Grammar Closure Under Union Operation

Central to the idea of composing smaller grammars to create larger ones is the concept of grammar *union*, because a larger grammar can be treated simply a union of smaller ones. It is well known that context free grammars (CFGs) are closed under the union operation. A proof of this fact can be found in most textbooks on formal languages[3]. This result provides a theoretical basis for creating context free grammar modules that are composable into larger grammars [9].

2.1.6 Attributed Context Free Grammars

LingoLogic grammars, however, allow for more expressiveness than would normally be the case with a standard CFG due to the use of attributes. Attributes are extra values attached to the terminals in the form of name-value pairs. The values can then be used in the grammar rules to add constraints to rules. It is these constraints that give LingoLogic its expressive power [10]. For example, suppose that the nonterminals in Example 1 had an extra value called *number* associated with them. The nonterminals might be re-written as follows[4]:

noun ->DOG[number=singular] |

DOGS[number=plural] |

CAT[number=singular] |

CATS[number=plural]

verb -> EATS[number=singular] |

EAT[number=plural] |

CHASES[number=singular] |

CHASE[number=plural]

A constraint could then be added to the nounPhrase and verbPhrase rules so these nonterminals adopt the number attribute of their terminals[5].

nounPhrase -> article noun

[nounPhrase.number = noun.number]

verbPhrase -> verb nounPhrase

[verbPhrase.number = noun.number]

Finally, a constraint to the top level rule could specify that the number of the noun-phrase and verb-phrase must agree.

S -> nounPhrase verbPhrase

[nounPhrase.number == verbPhrase.number]

Example 2: A simple attributed English grammar

Adding these attributes allow the parser to distinguish between singular and plural nouns and verbs, which means that sentences such as "Dogs eat cats" are allowed, while sentences such as "Dogs eats cat" are not.

2.2 *Translations*

The LingoLogic parser also has the ability to translate a sentence into a target language [10]. Translations can take many different forms. In the case of relational databases, the natural language queries that are formed via the cascading menus are translated into SQL. The agent system created by the EiA project requires its messages to be XML, so translation rules could be written to build well-formed XML messages. A third idea for a target language is a function call that could then be evaluated by another program. An example of this would be the sentence "Microwave cook for 30 seconds" could be translated to a call to a *microwave* method with arguments *cook* and *30 seconds*. It might look like this:

```
Microwave(cook,30);
```

2.3 *Advantages of Distributed Grammars*

In general, distributed systems have several advantages over stand-alone type systems. The first is that distributed systems can be more robust in that they eliminate single points-of-failure. For example, in a packet switched network, such as the Internet, if a router crashes, packets are still able to reach their destination via another route [11]. In the same way, if a very large grammar file is broken into smaller pieces, these pieces could be stored redundantly in different places, allowing the entire system to function, even if a failure occurs with one piece of the grammar. Secondly, distributed systems can be more efficient because data that is not relevant does not need to be processed. In terms of a distributed grammar, smaller grammar files could be downloaded and composed on the fly, eliminating the need to process rules and lexicons that will not be used. Finally, distributed systems are scalable. This characteristic is important and is one of the driving factors for creating grammar modules in the first place, because it will allow for grammars to be written for agents as they are made. This means there is no need to update a central grammar each time a new agent is added to the system; rather, when a new agent is added, its grammar will be an extension and processed as it is needed.

2.4 *Related Work*

The idea of adding the ability for grammars to be composed of smaller grammars in order to allow for them to be distributed is not new. Because of the obvious benefits of flexibility and robustness that would be gained from this capability, distributing grammar files was a central concept in OBJS' quest to enable agents to be controlled via an MBNLI on the Web.

2.4.1 *AgentGram*

AgentGram was a prototype of a MBNLI system that was developed from 1998-1999. Unlike LingoLogic, it did not parse context-free languages; instead it handled only a simpler tree-grammar and did not handle translations. AgentGram did, however, have the ability to load grammar files on-the-fly, thus allowing grammars to be chained together. When a terminal or nonterminal that was not specified in the current grammar file was reached, a URL pointed to the grammar file that could complete the rule. This URL was then followed, the grammar downloaded and processed, and the parse continued seamlessly [12]. For example:

```
<item name = "List the">
   <item name = "hotels">
      <item name = "in L.A.">
         <item name = "[THEN]"
URL  = "http://...hotels.xml">
         <item name = "in Washington D.C.">
            <item name = "[THEN]"
URL ="http://...hotels.xml">
```

The file hotels.xml would then have some information specific to hotels. It might look something like this:

```
<item name = "where the hotel name is">
   <item name = "Best Western">
   <item name = "Clarion">
<item name = "where the hotel costs">
   <item name = "less than $70 per night">
   <item name = "between $71 and $100 per night">
   <item name = "more than $101 per night">
```

Example 3: An AgentGram XML grammar

This example should clarify the advantages of a system of distributed grammars. The MBNLI system has to load only relevant grammar rules when it requires them instead of loading all rules that it might ever use. Furthermore, this distributed type of system is easier to scale and maintain because, when the system changes, only relevant files must be updated. For example, if a new hotel were built in L.A. only the hotel grammar would need to be modified.

2.4.2 *Patent Application*

The idea for a system of distributed grammars was conceived for the LingoLogic MBNLI system as well, though it was not implemented. An OBJS patent application [13] describes a method of "chaining" grammars together by encapsulating the grammars (productions, terminals, etc.) into grammar descriptors. Besides containing the grammar, the descriptors would include other information as well, such as a set of pointers to other descriptors that the grammar references. The parser can either process the references actively by recursively processing the descriptors that the higher level descriptors point to until there are no more links, or lazily by processing a descriptor only when needed.

3 Approach:

A multi-agent system will contain many agents of various kinds – various human roles as well as vehicles, equipment, pets, sensors, and even passive things like pictures. Assume that each agent can have an associated grammar. In order to scale a multi-agent system, agents can come and go so it must be possible to add or remove grammars dynamically. This means that grammars will have to be loaded on the fly and that grammars of the system will need to be able to be composed dynamically. There are various ways to do this. The LingoLogic patent application [13] describes one way involving grammar descriptors and chaining grammars together. In spite of a limitation of the current implementation of LingoLogic (that all nonterminals for a grammar must be known before any rules are specified), we can still simulate breaking up the grammar files for a system into grammar modules.

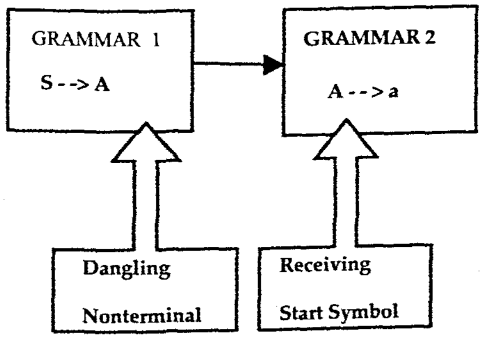
3.1 *Context Free Grammar Modules*

In their most basic form, CFG Modules are simply CFGs that have been broken into semantic groups. Each group is a CFG in its own right, meaning that it has a start symbol, a set of nonterminals, a set of terminals, and a set of productions. The difference is that a nonterminal in the RHS of a higher level grammar will link to the start symbol of a lower-level CFG module, enabling the chaining of several smaller grammars into a larger one.

3.1.1 *Terminology*

For the sake of clarity, two terms will be defined. A *dangling nonterminal* is the RHS nonterminal in a higher level grammar that links it to the start symbol of a lower level grammar. A *receiving start-symbol* is the LHS nonterminal of a lower level grammar to which a dangling nonterminals points.

Figure 1: A dangling nonterminal and a receiving start symbol



3.1.2 *Types of CFG modules*

When using CFG modules as a way to distribute agent MBNLI grammars, there are two possible reasons for creating a certain module. The first is that a "plug-in" ability is desired. This is achieved when one dangling nonterminal points to several receiving start symbols. The second is when module reuse is desired. This is achieved when several dangling nonterminals point to one receiving start symbol. Although these two types of CFG modules are not mutually exclusive, both types will be examined distinctly.

One-to-many modules

In order to scale the NLI system to many agents, it will need to be easy to add or "plug in" an agent to the system. CFG modules provide this feature naturally. In this case, a higher level dangling nonterminal points to the receiving start symbols of several agent grammars.

```
S      - > AGENT

AGENT - > agent A

AGENT - > agent B
```

Many-to-one modules

Sometimes several agents will share similar features. In this case, an ability to reuse grammar files is desired. CFG modules support this scenario by allowing several dangling nonterminals to point to the same receiving start symbol.

```
AGENT _ A      -> FEATURE _ 1

AGENT _ B      -> FEATURE _ 1

FEATURE _ 1    -> foo bar
```

3.2 *Smart House: An Example*

The idea behind CFG Grammar modules will become clear with an example. A smart house example will be used. Imagine a house full of normal objects that each have a wrapper that allows them to be identified as unique objects by a computer and can be controlled via devices that downloads the object's grammar. The complete grammar of the house might look like this:

```
S - > microwave MICROWAVECOMMAND
    | oven OVENCOMMAND
    | thermostat THRMSTCOMMAND
```

```
MICROWAVECOMMAND - > turn on
                     | turn off
                     | time for TIME
                     | cook for TIME

OVENCOMMAND - > turn on
                | turn off
                | time for TIME
                | cook for TIME
                | preheat until temp is
                  PREHEAT-TEMP

THRMSTCOMMAND - > turn on heat in ROOMS
                  | turn on heat in ROOMS until
temp is ROOM-TEMP
                  | turn on air in ROOMS
                  | turn on air in ROOMS until
temp is ROOM-TEMP
                  | turn off heat in ROOMS
                  | turn off heat in ROOMS until
temp is ROOM-TEMP
                  | turn off air in ROOMS
                  | turn off air in ROOMS until
temp is ROOM-TEMP

TIME - > 30s | 1m | 2m | 3m | 4m | 10m

PREHEAT -> TEMP - > 300 | 325 | 350 | 374 | 400 |
450

ROOMS -> ROOM
       | ROOM and ROOMS

ROOM -> living room | dining room | master
bedroom | kitchen

ROOM -> TEMP - > 60 | 65 | 70 | 75 | 80
```

Example 4: The smart house grammar

3.3 *Forming CFG Modules*

This grammar can be divided into semantic groups, and the groups can be represented as files as follows:

```
******************************

MAIN GRAMMAR

S - > DEVICE-AND-COMMAND

******************************

MICROWAVE GRAMMAR

DEVICE-AND-COMMAND -> microwave
MICROWAVECOMMAND

MICROWAVECOMMAND - > POWER | TIMER |
COOK

******************************

OVEN GRAMMAR

DEVICE-AND-COMMAND -> oven
OVENCOMMAND

OVENCOMMAND - > POWER | TIMER | COOK |
PREHEAT

******************************

THERMOSTAT GRAMMAR

DEVICE-AND-COMMAND -> thermostat
THRMSTCOMMAND

THRMSTCOMMAND -> POWER heat in ROOMS
                 | POWER air in ROOMS
                 | POWER heat in ROOMS
   until temp is ROOM-TEMP
                 | POWER air in ROOMS
   until temp is ROOM-TEMP
                 | display_temperature in
   ROOM

******************************

POWER GRAMMAR

POWER - > turn on | turn off

******************************

TIMER GRAMMAR

TIMER - > time for TIME

******************************

COOK GRAMMAR

COOK - > cook | cook for TIME

******************************

TIME GRAMMAR

TIME - > 30s | 1m | 2m | 3m | 4m | 10m

******************************

PREHEAT GRAMMAR

PREHEAT - > preheat until temp is PREHEAT-TEMP

PREHEAT-TEMP - > 300 | 325 | 350 | 375 | 400 | 450
```

```
******************************

ROOMS GRAMMAR

ROOMS - > ROOM and ROOMS

ROOM - > living room | dining room | master
bedroom | kitchen

******************************

ROOM-TEMP GRAMMAR

ROOM-TEMP - > 60 | 65 | 70 | 75 | 80

******************************
```

Example 5: Smart house grammar modules

In this set of example grammar modules, it is easy to see that both the one-to-many and many-to-one module paradigms are used. The one-to-many type modules are used to add new devices to the system. This is possible because the dangling nonterminal in the highest level module is *DEVICE-AND-COMMAND*; adding a new device to the system relatively simple: just make the receiving start symbol of the new device grammar *DEVICE-AND-COMMAND*. Similarly, two modules sharing a module, utilizing the many-to-one feature of CFG modules, is also exemplified in the smart house set of grammar modules. The microwave grammar and the oven grammar both share the cook grammar module. They do this by have a dangling nonterminal called *cook*. The cook module's receiving start symbol is also named *cook*.

3.4 *Limitations of Grammar Modules*

Although the simple conversion of a CFG to a collection of CFG modules addresses several issues that are important to scaling a multi-agent system, it has several inherent limitations. One of the major limitations of grammar modules is that all the modules are global because there is no inherent scoping mechanism. This limitation creates two problems. The first is that there may be situations in which certain users should not be able to control certain agents. The second problem is that certain modules cannot be reused despite the fact that they share similar features. A third problem (unrelated to the scoping problems) with CFG modules is that many times a large (or even infinite) number of nonterminals should be able to be used, but due to the fact that it is difficult to specify ranges of values in CFGs, presenting the user with a list of all possible values from which to choose is not practical.

3.4.1 *Security*

In certain situations, certain users should not have access to every feature of an entire system. This concept is familiar in the world of operating systems, in which only administrators can control certain programs. Similarly, perhaps a parent does not want a child to be able to control the *power* feature of a thermostat agent but would like him to be able to issue the *display_temperature* command. Because all grammar rules are global, a new thermostat grammar module would have to be written that excluded the thermostat rules that began with the nonterminal*power*. This new thermostat grammar module would have to be loaded instead of the other one when the child is using the thermostat. A better solution might be to add a way to scope grammar rules so that the *power* rules are not accessible to the child.

3.4.2 *Reusability of many-to-one modules*

In a related problem to that of the security issue, CFG modules can only be reused when exactly the same feature set is desired. For example, a new grammar module that allows a user to control lamps is created such that it has the following grammar:

```
******************************

DEVICE-AND-COMMAND - > POWER lamp in
ROOMS

******************************
```

This lamp grammar would control lamps in the exact same rooms in which the thermostat can control the temperature. Suppose, however, that there is no lamp in the kitchen. This means that a new *rooms* CFG module must be created that is the same as the old *rooms* module but does not include the kitchen as an available terminal. This inability to scope causes the need for a new grammar module to be created despite the fact that one that is very similar already exists. In a large scale system, the inflexibility caused by the lack of a scoping mechanism might undermine the benefits gained from the ability to reuse CFG modules.

3.4.3 *Inability to Specify a Value from a Range of Terminals*

Many times the need to be able to specify a value from a range is required. This limitation is illustrated in the smart house example: the *time* module allows only a small handful of values.

One way to solve this problem would be to list every possible value as a terminal. This method, however, would also be cumbersome not only from the grammar writer's point of view, but also from the user's point of view when he has to pick a value from a very large menu. Another possible solution would be to write another grammar for the time module such as the following:

```
TIME - > MIN SEC

MIN - > DIGIT DIGIT min

SEC - > DIGIT DIGIT sec

DIGIT - > 0|1|2|3|4|5|6|7|8|9
```

The problem is that this grammar allows values such as *2 min 88 sec*, which is not correct.

3.5 Using Attributes to Scope CFG Modules

In Section 2.1.6, the idea of adding attributes to CFGs in order to extend the expressiveness of them is discussed. Attributes, which are essentially trees consisting of name-value pairs in the context of LingoLogic, can be used in order to add a sort of scoping mechanism to CFG modules.

3.5.1 Using Attributes to Solve the Security Problem

In the previous security example, a parent would like their child only to be able to use the *display-temperature* command in the thermostat module. In order to use attributes to solve this problem, first assume that a global attribute called *user* was added to the system. The thermostat module could be re-written as follows:

```
******************************

THERMOSTAT GRAMMAR

DEVICE-AND-COMMAND - > thermostat
THRMSTCOMMAND

    [global.user ∈ THERMSTCOMMAND.users]

THRMSTCOMMAND - > POWER heat in ROOMS
    [users = {parent}]

    | POWER air in ROOMS [users = {parent}]

    | display_temperature in ROOM

    [users = {parent, child}]

******************************
```

Example 6: Using attributes to add security to the Thermostat grammar

Rewriting the thermostat grammar in this way, only allows the parent to access the *power* commands because the *DEVICE-AND-COMMAND* rule is only able to be used when the *user* attribute of a global attribute tree is a member of the *users* attribute tree of *THEMSTCOMMAND.*

3.5.2 Using Attributes to Increase Reusability of Modules

As mentioned above, a module can be reused only if more than one higher-level modules share the *exact* same feature set. The previously used example was a *lamp* grammar that shares many, but not all, of the same rooms as the *thermostat* grammar. The solution, to create a new *rooms* grammar that contains a subset of the rooms listed in the original grammar used by the thermostat, is not scalable because it would create a multitude of very similar grammars in a large scale system.

Attributes could be applied in a similar way to the solution to the security issue to solve this problem. If an *objsInRoom* attribute were added to each room in the *rooms* grammar, then the set of objects available to be controlled in each room could be specified.

```
******************************

ROOMS GRAMMAR

ROOMS -> ROOM and ROOMS

[ROOMS.objsInRoom = ROOM.objectsInRoom]

ROOM - > living room [objsInRoom = {thermostat,
    lamp}]

    | dining room [objsInRoom = {thermostat, lamp}]

    | master bedroom [objsInRoom = {thermostat,
lamp}]

    | kitchen [objsInRoom = {lamp}]

******************************
```

The thermostat and lamp grammar could be rewritten to test for membership in each of the rooms.

```
******************************

THERMOSTAT GRAMMAR

DEVICE-AND-COMMAND - > thermostat
THRMSTCOMMAND

THRMSTCOMMAND - > POWER heat in ROOMS

    [thermostat " ROOMS.objsInRoom]

    | POWER air in ROOMS

    [thermostat " ROOMS.objsInRoom]

    | POWER heat in ROOMS until temp is ROOM-
TEMP

    [thermostat " ROOMS.objsInRoom]

    | POWER air in ROOMS until temp is ROOM-
TEMP

    [thermostat " ROOMS.objsInRoom]

    | display_temperature in ROOM

    [thermostat " ROOMS.objsInRoom]

******************************

LAMP GRAMMAR

DEVICE-AND-COMMAND - > POWER lamp in
ROOMS

    [lamp " ROOMS.objsInRoom]

******************************
```

Example 7: Using attributes to increase the reusability of the Room grammar

In this way, attributes can make CFG modules reusable, even if only a subset of its rules or terminals is to be used.

3.6 Using Experts to Specify a Value from a Range of Terminals

It is difficult to specify ranges of values from which to choose a value using CFGs. One solution is to let terminals be either values (as they currently are) or function calls that return a value. The function could then have some logic that would allow a user to choose easily a value from a range. For example, the *time* grammar could be changed so that instead of the CFG rules used to specify a time, a function is called that executes code that allows a user to specify the time.

```
******************************

TIME GRAMMAR

TIME - > timer()

******************************

TIMER FUNCTION

/* Note that this function is for specifying a timer
time, such as "count for 4 hours and 20 minutes." It
is not for specifying a time of day like "4:00 PM". */

1 INT hour, min;

2 PRINT "ENTER HOURS";

3 GET hour;

4 IF (hour < 0) {PRINT "HOUR MUST BE >=0";

GOTO 2}

5 PRINT "ENTER MINUTES";

6 GET min;

7 IF (min > 60) { PRINT "MIN MUST BE < 60;

 GOTO 5}

8 IF (min < 0) { PRINT "MIN MUST BE >= 0; GOTO 5}

9 RETURN hour + ":" + min;

******************************
```

Example 8: Using an expert to specify a range of timer times

The functions that return values are called *experts* because they are an "expert" at knowing a particular field. It can be seen that experts can help a user choose a value from all kinds of ranges including integers, currency, and time of day. If the logic in the expert were more complicated than the simple *timer* function above, a graphical user interface could be used to specify the value. A useful example might be that the expert produces a color chart and allows the user to visually select a color [14].

4 Implementation:

4.1 LingoLogic Grammar files

The smart house grammar was implemented using the LingoLogic Toolkit. LingoLogic consists of a predictive parser and an interface. The interface lets the user choose a phrase from a set of choices, each representing continuations of the sentence, and then sends the phrase back to the parser which returns a set of next possible choices. This process is continued until a complete sentence is built. The LingoLogic parser is controlled via Lisp-like commands. A LingoLogic grammar has four parts: the parser initialization, category declaration, lexicon definitions, and rule definitions [10].

4.1.1 Parser initiation

The first step in writing a LingoLogic grammar is to initialize the parser. This involves the following statement, which instantiates a parser object and creates a pointer to the newly created parser object. The parser initialization statement has the following syntax, where *parser1* is the pointer to the parser:

```
(setq parser1 (create-parser))
```

4.1.2 Parser Categories

Secondly, the parser categories are defined through a call to the function *set-parser-categories*. Parser categories are the set of nonterminals that will be used in the grammar. In other words, no nonterminal may be used in the grammar rule unless it was declared to the parser. LingoLogic requires that all parser categories be declared before the first rule is defined. The function *set-parser-categories* may, however, be called multiple times, as long as the final time that it is called is before the first grammar rule is declared. The final set of parser categories is the union of the parser categories declared in each call to *set-parser-categories*. The function *set-parser-categories* has the following syntax, where'*parser1* is a pointer to the parser.

```
(set-parser-categories parser1 '(<category symbols>))
```

4.1.3 Lexicon

The lexicon is the set of terminals used in a certain grammar. Terminals are defined using a call to the function *defword*, with the following syntax, where *word1* is the name of the word being defined, *parser1* is a pointer to the parser, *<attribute-tree>* is a tree of attributes associated with the word, *<menu>* is the menu on which the word will appear, *<print-info>* is the string that will appear on the menu, and *[expert-info]* is an optional function that may be run to aid the user in specifying a value [15]:

```
(DEFWORD word1 parser1 <attribute-tree>

(<menu> <print-info> [<expert-info>]))
```

The most complex aspect of this definition is the *<attribute-tree>*. As previously mentioned in Section 2.1.6, LingoLogic has the ability to parse attributed grammars. Attributes are extra values attached to terminals that add semantics not imparted by the syntax of the grammar [16]. In the context of LingoLogic, the attributes are defined as name-value pairs or name-value lists that form attribute trees. Constraints can then be added in the form of rules that use the trees to add expressive power. One attribute, *:cat*, short for category, is required by the parser, because it associates the word with a parser category. The lexicon for the attribute English grammar of Example 2, expressed using calls to *defrule* would look as follows:

(defword dog p1 (((:cat noun)(number sing)) nouns "dog"))

(defword dogs p1 (((:cat noun)(number plural)) nouns "dogs"))

(defword cat p1 (((:cat noun)(number sing)) nouns'"cat"))

(defword cats p1 (((:cat noun)(number plural)) nouns "cats"))

(defword chases p1(((:cat verb)(number sing)) verbs "chases"))

(defword chase p1(((:cat verb)(number plural)) verbs "chase"))

(defword eats p1 (((:cat verb)(number sing)) verbs "eats"))

(defword eat p1 (((:cat verb)(number plural)) verbs "eat"))

4.1.4 *Grammar Rules*

The most complex portion of a LingoLogic grammar file is the set of grammar rules. Like all other LingoLogic parser commands, the grammar rules are written using a LISP-like syntax[6]. The rules are defined using the following syntax, where *rule1* is the name of the rule, *parser1* is the pointer to the parser, *<term1>* is the LHS, *<term2>*... is the RHS that define *<term1>*, and *[<constraint>]* are optional constraints that may be added due to the ability of the parser to handle attributes [10].

(defrule rule1 parser1 (<term1> —> <term2> <term3>...)

[<constraint> ...])

In Example 1, a simple English language grammar is introduced. If this grammar were written for LingoLogic, the top level rule would be defined as follows:

(defrule rule1 parser1 (S—> nounPhrase verbPhrase))

Example 2 augments the simple grammar with attributes. Grammar rules can then use these attributes in constraints that add expressive power. For example, a constraint which requires the number of the *nounPhrase* and *verbPhrase* to agree can be added to the rule above.

(defrule rule1 parser1 (S—> nounPhrase VerbPhrase)

((nounPhrase number) = (verbPhrase number)))

4.2 *LingoLogic CFG Modules*

The first step in implementing CFG modules in LingoLogic is to write the grammar rules and break them into conceptual groups, as in Section 3.3. Then the grammar rules must be translated to the LISP-syntax of the LingoLogic parser command set. Next, the parser category declarations have to be separated into their own files. Finally, a parser initialization file must be created.

4.2.1 *Writing LingoLogic Grammar Files*

The translation of the CFG production rules into LingoLogic grammar functions consists of writing a *defrule* function for each definition of every nonterminal in the grammar. In other words, for each nonterminal α in grammar *G*, a *defrule* function must be written for every β where $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \ldots \beta_n$. For example, the microwave module has a production rule that looks like this:

MICROWAVECOMMAND -> POWER | TIMER | COOK

The translation of this rule into a LingoLogic grammar rule involves writing three *defrule* functions.

(defrule rule1 parser1 (MICROWAVECOMMAND — > POWER))

(defrule ruel2 parser1 (MICROWAVECOMMAND — > TIMER))

(defrule rule3 parser1 (MICROWAVECOMMAND — > COOK))

Furthermore, a *defword* function must be written for every terminal in the grammar. The *:cat* attribute of the word must the be nonterminal which is defined by that word. For example, the room module has a production rule that looks like this:

ROOM -> living room | dining room | master bedroom | kitchen

In order to write the LingoLogic translation of this rule, a *defword* function must be written for each room.

(defword livingroom parser1 (((:cat room)) rooms "Living Room"))

(defword diningroom parser1 (((:cat room)) rooms "Dining Room"))

(defword mbedroom parser1 (((:cat room)) rooms "Master Bedroom"))

```
(defword kitchen parser1 (((:cat room)) rooms
"Kitchen"))
```

One important detail of LingoLogic *defrule* function is that the RHS of the productions may consist only of pointers to parser categories, which implies that the RHS of a production rule may *not* contain a pointer to a *defword* function. This means that terminals are included in grammar rules indirectly by means of the parser category (:cat) attribute of the *defword* function.

A final aspect of translating a CFG into LingoLogic grammar functions is the rule namespace. Each *defrule* and *defword* function is global, so care must be taken not to give two rules or two words the same name.

4.2.2 Creating the Parser Category Files

The LingoLogic parser requires that all parser categories be declared before the first grammar rule is defined. This requirement is a result of the way the parser handles the construction of some internal data structures. Therefore, for each grammar module, a parser category file must be created. This file consists only of a call to *set-parser-categories* (see Section 4.1.2) and declares parser categories on the RHS of every grammar rule in the file. In this way, the parser can first execute all of the *set-parser-categories* functions by reading all of the parser category files before it reads any of the *defrule* or *defword* functions.

4.2.3 Parser Initialization File

The final file that must be created is a parser initialization file, which has three parts. The first part is to create the parser with a command such as the following:

```
(setq parser1 (create-parser))
```

The second step is to load all of the parser category files.

```
(load "main.pc") ; ; load main parser categories.

(load "microwave.pc") ; ; load microwave parser
categories

(load "oven.pc") ; ; load oven parser categories
```

...The third step is to load all of the grammar files.

```
(load "main.gnl") ; ; load the main grammar

(load "microwave.gnl") ; ; load the microwave
grammar

(load "oven.gnl") ; ; load the oven grammar
```

Finally, a batch file is created that starts the parser and initializes it with the initialization file and starts the front end.

5 Conclusions:

5.1 Significance

As more and more objects are added to the Everything is Alive agent system project, a standardized means to control them will be needed. LingoLogic, due to its intuitive, guided approach to issuing commands is a good candidate for the mechanism to control agents. In order for LingoLogic to be useful, however, the agent grammars must be able to scale. Context Free Grammar modules provide a means to create scalable agent grammars that can be composed together at run time to create a menu based natural language interface to the agent system.

5.2 Future Work

One area of improvement for this project involves the binding time of the parser. Because the parser requires all parser categories to be declared before the first grammar rule is defined, grammar rules that involve new categories cannot be added after the parser has been initialized. This means that grammars cannot be added on the fly. It would be better if the parser were improved to accept new parser categories even after grammar rules have been declared. Then, grammars could be truly dynamic, in that new modules could be added on the fly.

Along with the ability to add new grammars on the fly, it would be useful if grammar descriptors, as described in [13] were developed. Grammar descriptors would provide a better encapsulation model for LingoLogic grammars, because a descriptor would be meta information at the top of the grammar file that describes how it is chained to other grammars.

Finally, another area of improvement to this project would be to add the ability to detect which agents are connected to the agent system, allow the user to select which agents he would like to communicate with, then pull the selected agents' grammar modules from a database. Such a system would simulate how a soft controller might work in the future.

6 References:

[1] C. Thompson, "Everything is Alive," Architectural Perspective Column, *IEEE Internet Computing*, Jan-Feb 2004. http://csce.uark.edu/~cwt/DOCS/2004-01—PAPER—IEEE-Internet-Computing—Everything-is-Alive.pdf

[2] C. Thompson, "Smart Devices and Soft Controllers," Architectural Perspective Column, *IEEE Internet Computing*, Jan-Feb 2005. http://csce.uark.edu/~cwt/DOCS/2005-01—PAPER—IEEE-Internet-Computing—Smart-Objects-and-Soft-Controllers.pdf

[3] Vu, Minh, Craig Thompson, "E2 Agent Plugin Architecture," *IEEE International Conference on Integration of Knowledge Intensive Multi-Agent Systems* (KIMAS-05), Waltham, MA, April 18-21, 2005. http://csce.uark.edu/~cwt/DOCS/2005-01—PAPER—KIMAS-05—Vu-Thompson—E2-agent-plugin-architecture.doc

[4] J. Robertson, C. Thompson, "Everything is Alive Agent Architecture," *IEEE International Conference on Integration of Knowledge Intensive Multi-Agent Systems* (KIMAS-05), Waltham, MA, April 18-21, 2005. http://csce.uark.edu/%7Ecwt/DOCS/2005-01—PAPER—KIMAS-05—Robertson-Thompson—Everything-is-Alive-Agent-System.doc

[5] J. Hoag, C. Thompson, "RFID Agent Middleware Architecture,"

Conference on Applied Research in Information Technology, Conway, AR, March 3, 2006. http://csce.uark.edu/~cwt/COURSES/2006-01—CSCE-490-590—RFID-Agent-Middleware/DOCS/2005-12—PAPER—ALAR—RFID-Agent-Middleware—Hoag-Thompson—long.doc

[6] C. Thompson, P. Pazandak, "Introduction to Menu-based Natural Language Interfaces," Technical Memo, Object Services and Consulting Inc., 2000, http://www.objs.com/agility/tech-reports/0101-MBNLI.doc

[7] C. Thompson, G. Hansen, "NLI Query Interface," Object Services and Consulting Inc. September 1998, http://www.objs.com/OSA/NLI-Query-Service.html

[8] C. Thompson, "Talk to your Semantic Web," Architectural Perspective Column, *IEEE Internet Computing*, Nov-Dec 2005.

[9] J. Hopcroft, R. Motwani, J. Ullman, Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, 2001.

[10] G. Lystad and R. Roberts, Chapter 5 in "LingoLogic Manual," Object Services and Consulting Inc., 2000.

[11] J. Kurose, K. Ross, Computer Networking, United States of America, Pearson Education, 2005

[12] P. Pazandak, C. Thompson, "AgentGram: Natural Language Interface for Agents," Project Summary, Object Services and Consulting Inc. June 2002, http://www.objs.com/agility/final/AgentGram/AGENTGRAM-PROJECT-SUMMARY.html

[13] P. Pazandak, C. Thompson, "Guided Natural Language Interface System and Method." Patent Application, August 2000.

[14] G. Lystad and R. Roberts, Chapter 9 in "LingoLogic Manual," Object Services and Consulting Inc., 2000.

[15] G. Lystad and R. Roberts, Chapter 4 in "LingoLogic Manual," Object Services and Consulting., 2000.

[16] "Attribute Grammar," Wikipedia.com, http://en.wikipedia.org/wiki/Attribute_grammar

Endnotes:

[1] Although these examples may seem trite, imagine the benefits of seamless object interaction in a hospital or an airport.

[2] Please note the convention that nonterminals are lowercase and terminals are upper-case

[3] See Ullman, pg. 284 for one example of this proof.

[4] By convention, brackets indicate an attribute specification.

[5] Note the convention that the dot indicates selection, the equal sign indicates an assignment and two equal signs indicate a test for equality.

[6] The LISP-syntax of the parser command set is an artifact of the fact that LingoLogic's predecessor, NLMenu, was developed at Texas Instruments on LISP Machines in the 1980s. Although LISP is not as popular today, its advantages should not be overlooked: LISP's ability to manipulate symbols accounts for the resemblance of grammar rule definitions to the traditional CFG arrow notation in the formal languages literature.

Faculty comments:

Dr. Craig Thompson, Mr. Neumeier's mentor, made the following comments about his student's work:

First, a few words about Kyle – as it happens, this afternoon, Kyle is receiving an award as the Top Undergraduate Senior in Computer Science from the CSCE Department in the annual College of Engineering end of year meeting. This follows on the heels of Kyle providing technical support for a Walton School of Business entrepreneurial team that placed in the top ten out of 100 in the U San Francisco Business Plan Competition held over Spring Break, beating out MIT, Harvard and other top schools, and, more recently, also placing second in the Arkansas Governor's Cup Business Plan Competition and winning the Technology Award (based on the technology described below). Finally, Kyle has served informally as TA in my graduate class on Natural Language Interfaces this semester.

Now, about Kyle's work: Over the last year, Kyle received two undergraduate research grants to work with me on menu-based natural language on the topic of grammar composition. Imagine a collection of software systems (called agents) that can communicate with each other. Each one might represent and control a different device or thing in a home, office, or anywhere (cars, robots, sensors, thermostats, pictures, Ö). Imagine each has an RFID tag (which is like a barcode that can be read at a distance). Now imagine when a person points a next-generation truly "universal remote" at these things, a grammar for controlling that thing is automatically downloaded into the universal remote. This would allow the person to communicate or control things in an unprecedented manner. Add to this that the grammar is in a sort of domain-restricted English and uses menus so a user always knows what they can talk to things about, even if they have never seen that particular thing or kind of thing before. Kyle's thesis is focused on the part of this vision that involves downloading the grammars and composing grammars so the user of the universal remote can control multiple things at once. This is a significant step towards pervasive computing where computing is embedded in the world around us, not just on our desktops.