A Computational Psycholinguistic Model of Natural Language Processing

Jerry T. Ball

Air Force Research Laboratory
6030 South Kent Street, Mesa, Arizona 85212
Jerry.Ball@mesa.afmc.af.mil
www.DoubleRTheory.com

Abstract
Double R Model (Referential and Relational model) is a computational psycholinguistic model of NLP founded on the principles of Cognitive Linguistics and implemented using the ACT-R cognitive architecture and modeling environment. Despite its psycholinguistic basis, Double R Model is intended for use in the development of large-scale, functional language comprehension systems. Given the inherently human nature of language, this approach may be reasonable from an AI as well as a psycholinguistic point of view.

Introduction
Communities working to implement NLP systems have progressed in improving system quality, particularly in limited domains. However, NLP systems still fall well short of human-level performance. Symbolic systems are still too brittle; statistical systems are too mushy. In part, this failure is due to the lack of an adequate linguistic theory on which to base such systems. Mainstream Generative Linguistic theory, with its focus on syntax in isolation from cognition and cognitive processing, is inadequate as a basis for the development of NLP systems. Model Theoretic Semantics attempts to redress the syntactic focus of Generative Linguistics, but also falls short as a theory of cognition and cognitive processing (Jackendoff 1983, 2002). The advent of Cognitive Linguistics (Langacker 1987, 1991; Talmy 2003), a linguistic theory with an explicit focus on cognition and cognitive processing, offers the prospect for the development of NLP systems with a sound linguistic basis. Further, the recent, general availability of ACT-R (Anderson & Lebiere 1998), a psychologically vetted cognitive modeling environment, facilitates the development of cognitively plausible NLP systems. Whether or not a psychologically motivated NLP system can improve on non-psychologically based systems is yet to be shown, but given the historically poor performance of most pure AI systems, it is a goal worth pursuing.

This paper provides an introduction to the theoretical basis of Double R Model, describes some initial design elements and their implementation, and discusses some advantages over alternative approaches. It continues with a processing example and concludes with a consideration of future research.

Theoretical Background
Double R Grammar (Ball 2004) is the Cognitive Linguistic theory underlying Double R Model. In Cognitive Linguistics, all grammatical elements have a semantic basis, including parts of speech, grammatical markers, phrases and clauses. Our understanding of language is embodied and based on experience in the world (Lakoff & Johnson, 1980). Categorization is a key element of linguistic knowledge, and categories are seldom absolute—exhibiting, instead, effects of prototypicality, radial structure and the like (Lakoff 1987). Our linguistic capabilities derive from basic cognitive capabilities—there is no autonomous syntactic component separate from the rest of cognition. Knowledge of language is for the most part learned and not innate. Abstract linguistic categories (e.g. noun, verb, nominal, clause) are learned on the basis of experience with multiple instances of words and expressions which are members of these categories, with the categories being abstracted and generalized from experience. Also learned are schemas which abstract away from the relationships between linguistic categories. Over the course of a lifetime, humans acquire a large stock of schemas at multiple levels of abstraction and generalization representing knowledge of language and supporting language comprehension.

Two key dimensions of meaning that get grammatically encoded are referential and relational meaning. Consider the expressions

1. The book on the table
2. The book is on the table

These two expressions have essentially the same relational meaning. They both express the relation “on” existing between “a book” and “a table”. However, their referential meaning is significantly different. The first expression, as a whole, refers to an object whereas the second expression refers to a situation. In the first expression the word “book” indicates the type of object being referred to and functions as the head of the expression, and the word “the” indicates that an object is being referred to and functions as a (object) specifier. The phrase “on the table” refers to a location with respect to which the object “the book” can be identified and functions as a modifier. In the second expression, the word “on” indicates the type of situation and functions as the head with the word “is” indicating that a current situation is being
referred to and functioning as a (situation) specifier. As the examples show, the joint encoding of referential and relational meaning leads to representations that simultaneously reflect both these important dimensions of meaning, with trade-offs occurring where the encoding of referential and relational meaning compete for expression.

Double R Process is the theory of language processing underlying Double R Model. It is a highly interactive theory of language processing. Representations of referential and relational meaning are constructed directly from input texts. Unlike many other theories, there is no separate syntactic analysis that feeds a semantic interpretation component. The processing mechanism is driven by the input text in a largely bottom-up, lexically driven manner. There is no top-down assumption that a privileged linguistic constituent like the sentence will occur. There is no phrase structure grammar and no top-down control mechanism. How then are representations of input text constructed? Operating on the text from left to right, schemas corresponding to lexical items are activated. For those lexical items which are relational or referential in nature, associated schemas establish expectations which both determine the possible structures and drive the processing mechanism.

Modern cognitive research (e.g. Kintsch 1998) has shown that a short-term working memory (STWM) is available to support cognitive processing. In Double R Process, this STWM is used for storing arguments which have yet to be integrated into a relational or referential schema, partially instantiated schemas, and completed schemas. If a relational or referential entity is encountered which expects to find an argument to its left in the input text then that argument is assumed to be available in STWM. If the relational or referential entity expects to find an argument to its right, then the entity is stored in STWM as a partially completed schema and waits for the occurrence of the appropriate argument. When that argument is encountered it is instantiated into the stored relational or referential schema. Instantiated arguments are not separately available in STWM. This keeps the number of separate linguistic units which must be maintained in STWM to a minimum.

**ACT-R**

ACT-R is a cognitive architecture and modeling environment for the development of computational cognitive models (Anderson & Lebiere 1998). It is a psychologically based cognitive architecture which has been used extensively in the modeling of higher-level cognitive processes, albeit on a smaller scale than that typical of AI systems. ACT-R is a hybrid system that includes symbolic production and declarative memory systems integrated with sub-symbolic production selection and spreading activation and decay processes. Production selection involves the parallel matching of the left-hand side of all productions against a collection of buffers (e.g. goal buffer, retrieval buffer) which contain the active contents of memory and perception. Production execution is a serial process—only one production is executed at a time. The parallel spreading activation process determines which declarative memory chunk is put in the retrieval buffer. ACT-R supports single inheritance of declarative memory chunks and limited, variable-based pattern matching. Version 5 of ACT-R (Anderson et al., 2002) adds a perceptual-motor component supporting the development of embodied cognitive models.

**Double R Model**

Double R Model is the implementation of Double R Theory within the ACT-R architecture. The major elements of the model are a defined lexical hierarchy, left-to-right input style, type inheritance, a chunk stack for short-term memory processing, context accommodation, limited backtracking, and dynamic functional assignment.

Double R Model is currently capable of processing an interesting range of grammatical constructions including: 1) intransitive, transitive and ditransitive verbs; 2) verbs taking clausal complements; 3) predicate nominals, predicate adjectives and predicate prepositions; 4) conjunctions of numerous grammatical types; 5) modification by attributive adjectives, prepositional phrases and adverbs, etc. Double R Model accepts as input as little as a single word or as much as an entire chunk of discourse, using the perceptual component of ACT-R to read words from a text window. Unrecognized words are simply ignored. Unrecognized grammatical forms result in partially analyzed text, not failure. The output of the model is a collection of declarative memory chunks that represent the referential and relational meaning of the input text.

**Inheritance vs. Unification**

Unification allows for the unbounded, recursive matching of two logical representations. Unification is an extremely powerful pattern matching technique used in many language processing systems based on Prolog. Unfortunately, it is psychologically too powerful. For example, the following two logical expressions can be unified:

\[ p(a,B,c(d,e,f(g,h(i,j),K),l)) \]
\[ p(X,b,c(Y,c,e,f(Z,T,U),l)) \]

where capitalized letters are variables and lowercase letters are constants. Humans are unlikely to be able to retain the full extent of such unifications for subsequent processing.

On the other hand, although extremely powerful, unification does not support the matching of types to subtypes. Thus, if we have a verb type with intransitive and transitive verb subtypes, unification cannot unify a chunk of type verb with a chunk of type intransitive verb or transitive verb. Unification’s inability to match types to subtypes often results in a proliferation of rules (or conditions on rules) to handle the various combinations. With inheritance,
Inheritance supports the matching of two representations without requiring the recursive matching of their subparts (unlike unification) so long as the types of the two representations are compatible. Types are essentially an abstraction mechanism which makes it possible to ignore the detailed internal structure of representations when comparing them. Of course, there may be productions that do consider the internal structure, but types are useful here as well. Instead of having to fully elaborate the internal structure, types can be used to partially elaborate that structure providing a (limited) unification like capability where needed. In sum, inheritance and limited pattern matching provide a psychologically plausible alternative to a full unification capability.

To take advantage of inheritance, Double R Model incorporates a type hierarchy. Some relevant elements of the current hierarchy of types are shown below:

Lexical-type
  Noun
  Verb
  Adjective
  Preposition
  Determiner
  Auxiliary
Referential-type
  Head
 Specifier
  Modifier
Referring-expression-type
  Object-referring-expression
  Situation-referring-expression
Relation-type
  Argument
Predicate

The more specialized a production is, the more specialized the types of the chunks in the goal and retrieval buffers to which the production matches will need to be. The most general productions match a goal chunk whose type is top-type and ignore the retrieval buffer chunk.

The Context Chunk and Chunk Stack

The current ACT-R environment provides only the goal and retrieval buffers to store the partial products of language comprehension—although earlier versions provided a goal stack. The lack of a stack is particularly constraining, since a stack is the primary data structure for managing the (limited) recursion that occurs in language. There needs to be some mechanism for retrieving previously processed words and expressions from STWM in last-in/first-out order during processing (subject to various kinds of error that can occur in the retrieval process). A stack provides this (essentially error free) capability. It is expected that a capacity to maintain about 5 separate linguistic chunks in STWM is needed to handle most input—supporting at least one level of recursion (and perhaps two for the more gifted).

To overcome these architectural problems, Double R Model introduces a context chunk containing a bounded, circular stack of links to declarative memory. As chunks are stacked in the circular stack, if the number of chunks exceeds the limit of the stack, then new chunks replace the least recently stacked chunks (supporting at least one type of STWM error). The actual number of chunks allowed in the stack is specified by a global parameter. This parameter is settable to reflect individual differences in STWM capacity. Chunks cannot be directly used from the stack. Rather, the stack is used to provide a template for retrieving the chunk from declarative memory. Essentially, the chunk on the stack provides a link to the corresponding declarative memory chunk. Since the chunk must be retrieved from declarative memory before use, the spreading activation mechanism of ACT-R is not circumvented and retrieval errors are possible—unlike the goal stack of earlier versions of ACT-R which provided perfect memory for goals.

Lexical and Functional Entries

The lexical entries in the model provide a limited amount of information which is stored in the word and word-info chunks as defined by the following chunk-types:

(chunk-type word word-form word-marker)
(chunk-type word-info word-marker
  word-root word-type word-subtype
  word-morph-type)

The word-form slot of the word chunk contains the physical form of the word (represented as a string in ACT-R); the word-marker slot contains an abstraction of the physical form. The word-root slot contains the value of the root form of the word. The word-type slot contains the lexical type of the word and is used to convert a word-info chunk into a lexical-type chunk for subsequent processing. A word-subtype slot is provided as a workaround for the lack of multiple inheritance in ACT-R 5. The word-morph-type slot supports the encoding of additional grammatical information.

Sample lexical entries for a noun and verb are provided below:

(cow-wf isa word
  word-form "cow"
  word-marker cow)
Note that there is no indication of the functional roles (i.e., head, specifier, predicate, argument) that particular lexical items may fulfill. Following conversion of word-info chunks into lexical-type chunks (e.g., verb, adjective), functional roles are dynamically assigned by the productions that are executed during the processing of a piece of text. Since functional role chunks are dynamically created, only chunk-type definitions exist for functional categories prior to processing. As an example of a chunk-type definition for a functional category, consider the category pred-trans-verb (i.e., transitive verb functioning as a predicate) whose definition involves several hierarchically related chunk-types as shown below:

(chunk-type top-type head)
(chunk-type (rel-type (include top-type)))
(chunk-type (pred-type (include rel-type)) subj spec mod post-mod)
(chunk-type (pred-trans-verb (include pred-type)) obj)

The top-type chunk-type contains the single slot head. All types are subtypes of top-type and inherit the head slot. Rel-type is a subtype of top-type that doesn’t add any additional slots. Pred-type is a subtype of rel-type that adds the slots subj, spec, mod, and post-mod. It is when a relation is functioning as a predicate that these slots become relevant. Finally, Pred-trans-verb is a subtype of pred-type that adds the slot obj.

Default Rules and Assignment Productions

ACT-R’s inheritance mechanism can be combined with the production utility parameter which guides production selection, to establish default rules. Since all types extend a base type (top-type), using the base type as the value of the goal chunk in a production will cause the production to match any goal chunk. If the production is assigned a production utility value that is lower than competing productions, it will only be selected if no other production matches. A sample default production is shown below:

(p process-default--retrieve-prev-chunk
 =goal> ISA top-type
 =context> ISA context
 state process
 chunk-stack =chunk-stack
 =chunk-stack> ISA chunk-stack-chunk
 this-chunk =chunk
 prev-chunk =prev-chunk
 =>
 =context>
 state retrieve-prev-chunk
 chunk-stack =prev-chunk
 +retrieval> =chunk)
 (spp process-default--retrieve-prev-chunk :p 0.75)

In this structure, p identifies a production, process-default--retrieve-prev-chunk is the name of the production, =goal> identifies the goal chunk, =context> identifies a context chunk, ISA context is a chunk type, state is a chunk slot, process is a slot value, => separates the left-hand side from the right-hand side and variables are preceded by = as in =chunk. This default production causes the previous chunk to be retrieved from declarative memory (using the +retrieval> form) if no other production is selected. To make this production a default production, the production utility parameter is set using the spp (set production parameter) command to a value of 0.75 (the default value is 1.0).

The following functional assignment production creates an instance of a pred-trans-verb and provides initial values for the slots:

(p process-verb--assign-to-pred-trans-verb
 =goal> ISA verb
 head =verb
 subtype trans-verb
 =context> ISA context
 state assign-verb-to-pred-verb
 =>
 +goal> ISA pred-trans-verb
 subj none
 spec none
 mod none
 head =goal
 post-mod none
 obj none
 =context>
 state retrieve-prev-chunk)

In this production, a verb whose subtype slot has the value trans-verb is dynamically assigned the function of a pred-trans-verb for subsequent processing (where +goal> specifies the creation of a new goal chunk). The only slot of pred-trans-verb that is given a value other than none is the head slot whose value is set to be the old goal chunk (head =goal). This production has the effect of assigning a transitive verb the functional role of predicate (specialized as a transitive verb predicate).
Context Accommodation vs. Backtracking

Context accommodation is a mechanism for changing the function of an expression based on the context without backtracking. For example, when an auxiliary verb like “did” occurs it is likely functioning as a predicate (of situation) specifier as in “he did not run” where the predicate is “run” and “did not” provides the specification for that predicate. However, auxiliary verbs may also function as (abstract) predicates when they are followed by a noun phrase as in “he did it”. Determining the ultimate function of an auxiliary verb can only be made when the expression following the auxiliary is processed. In a backtracking system, if the auxiliary verb is initially determined to be functioning as a predicate specifier, then when the noun phrase “it” occurs, the system will backtrack and reanalyze the auxiliary verb, perhaps selecting the predicate function on backtracking. However, note that backtracking mechanisms typically lose the context that forced the backtracking. Thus, on backtracking to the auxiliary verb, the system has no knowledge of the subsequent occurrence of a noun phrase to indicate the use of the auxiliary verb as a predicate. Thus, without additional structures to retain the information the system cannot make an informed selection of a new function for the auxiliary.

A better alternative is to accommodate the function of the auxiliary verb in the context which forces that accommodation. In this approach, when the noun phrase “it” is processed and the auxiliary verb functioning as a predicate specifier is retrieved, the function of the auxiliary verb can be accommodated in the context of a subsequent noun phrase to be a predicate. Context accommodation avoids the need to backtrack in this case and allows the context to adjust the function of an expression just where that accommodation is supported by the context.

Processing Example

As an example of the processing of a piece of text and the creation of declarative memory chunks to represent the meaning of the text, consider the following text:

The dog lover is asleep

The processing of the word “the” results in the creation of the following declarative memory chunks:

\[
\begin{align*}
\text{Goal25} & \quad \text{isa} \text{ DETERMINER} \\
& \quad \text{head The} \\
\text{Goal26} & \quad \text{isa} \text{ OBJ-SPEC} \\
& \quad \text{head Goal25} \\
& \quad \text{mod None}
\end{align*}
\]

The first chunk, goal25, is a determiner whose head slot has the value The. This chunk represents the inherent part of speech of the word “the”. The second chunk, goal26, is an obj-spec (object specifier) whose head slot has the value goal25 and whose mod slot has the value none. This second chunk represents the referential function of “the”. Note that if “the” were the only word in the input text, the creation of these two chunks would still occur since the processing mechanism works bottom-up from the lexical items and makes no assumptions about what will occur independently of the lexical items.

The processing of the second word “dog” creates the following declarative memory chunks:

\[
\begin{align*}
\text{Goal39} & \quad \text{isa} \text{ NOUN} \\
& \quad \text{head Dog} \\
\text{Goal40} & \quad \text{isa} \text{ HEAD} \\
& \quad \text{head Goal39} \\
& \quad \text{mod None} \\
& \quad \text{post-mod None} \\
\text{Goal41} & \quad \text{isa} \text{ OBJ-REFER-EXPR} \\
& \quad \text{head Goal40} \\
& \quad \text{referent None-For-Now} \\
& \quad \text{spec Goal26} \\
& \quad \text{mod None} \\
& \quad \text{post-mod None}
\end{align*}
\]

Goal41 is a full object referring expression and contains a referent slot to support a link to an object in the situation model corresponding to this piece of text. Currently, the model cannot establish the value of the referent slot pending development of the situation model that will provide representations of the referents.

The processing of the third word “lover” creates or modifies the following declarative memory chunks:

\[
\begin{align*}
\text{Goal47} & \quad \text{isa} \text{ NOUN} \\
& \quad \text{head Lover} \\
\text{Goal48} & \quad \text{isa} \text{ HEAD} \\
& \quad \text{head Goal47} \\
& \quad \text{mod Goal40} \\
& \quad \text{post-mod None} \\
\text{Goal41} & \quad \text{isa} \text{ OBJ-REFER-EXPR} \\
& \quad \text{head Goal48} \\
& \quad \text{referent None-For-Now} \\
& \quad \text{spec Goal26} \\
& \quad \text{mod None} \\
& \quad \text{post-mod None}
\end{align*}
\]

Note that goal47 (“lover”) is now the head of goal48 which is the head of the object referring expression (goal41) with goal40 functioning as a modifier of goal48. This is an example of context accommodation.

Continuing with the next word “is” leads to the creation of the following chunks:

\[
\begin{align*}
\text{Goal54} & \quad \text{isa} \text{ REG-AUX} \\
& \quad \text{head Is-Aux} \\
\text{Goal55} & \quad \text{isa} \text{ PRED-SPEC} \\
& \quad \text{head Aux-1}
\end{align*}
\]
Note that the pred-spec chunk type has a modal-aux, neg, and three auxiliary slots (aux-1, aux-2, and aux-3) to handle the range of predicate specifiers that can occur.
For this instance of a pred-spec (goal55), goal54 fills the aux-1 slot and functions as the head of goal55.

The processing of the final word “asleep” creates the following declarative memory chunks:

```
Goal61
  isa ADJECTIVE
  head Asleep
Goal62
  isa PRED-ADJ
  head Goal61
  subj Goal41
  spec Goal55
  mod None
  post-mod None
```

In the context of the predicate specifier “is” (goal55), the adjective “asleep” functions as a predicate adjective filling the head slot of goal62 with goal41 (an object referring expression) filling the subj slot (subject).

Following the processing of “asleep” the model attempts to read the next word. The failure to read a word signals the end of processing and a wrap-up production is executed. This wrap-up production creates a situation referring expression (goal65) with goal62 filling the head slot.

```
Goal65
  isa SIT-REFER-EXPR
  head Goal62
  referent None-For-Now
  mod None
```

At the end of processing a single chunk of type situation-referring-expression is available in the chunk-stack to support subsequent processing with the determination of the referent slot pending development of the situation model.

**Summary and Future Research**

Double R Model may be the first attempt at the development of an NLP system founded on the principles of Cognitive Linguistics and implemented in the ACT-R cognitive modeling environment. Double R Model is a hybrid symbolic/sub-symbolic system which leverages the symbolic and sub-symbolic capabilities of ACT-R.

Much work remains to be done. Double R Model has not yet reached a scale at which it can handle more than a subset of English. To expand the sub-symbolic capabilities of Double R Model (e.g. in support of lexical disambiguation), we are evaluating the use of LSA, and considering improvements to ACT-R’s single-level spreading activation mechanism. Further, full implementation of referential meaning must await development of the situation model to represent the objects and situations that are referred to by input texts. Finally, it has not yet been demonstrated that Double R Model can achieve the level of success that has long eluded AI and computational linguistic based systems.

**Acknowledgements**

I would like to thank Kevin Gluck, Col (s) Stuart Rodgers and Wink Bennett of the Air Force Research Laboratory for supporting this research.

**References**


