Implementation of an agent-based model for studying the acquisition of language systems of logical constructions

by

Joan Ginés i Ametllé

Submitted to

Facultat d'Informàtica de Barcelona (FIB)
Universitat Politècnica de Catalunya (UPC) · BarcelonaTech

in partial fulfilment of the requirements for the

Bachelor's Degree in Informatics Engineering
Major in Computing

Director:

Dr. María Josefina Sierra Santibáñez
Department of Computer Science

1st July 2015

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

A la meva família i amics,

que d'alguna manera o altra m'ajuden
a arribar més lluny, sempre més lluny.

Abstract

Within the broad field of linguistics, the emergence and evolution of human languages remain matters without generally consensed theories. Nevertheless, and during recent years, the gradual introduction of agent-based models has provided a tool for hypothesis-validation which is accelerating the progress being made.

In this project we review some influential experiments in evolutionary linguistics and implement a basic form of an existing agent-based model for studying the acquisition of language systems of logical constructions. The model here presented has been built using the Prolog programming language and designed around a series of reusable modules, so that it can be employed by external researchers of related fields in conducting future studies.

Resum

Dins de l'ampli camp de la lingüística, l'origen i l'evolució dels llenguatges humans és manté com una qüestió sobre la qual no s'ha arribat a formular teories àmpliament acceptades. Tot i així, durant els últims anys, la introducció gradual de models basats en agents com a una eina per a validar hipòtesis ha accelerat notablement el progrés d'aquests estudis.

Aquest projecte analitza alguns experiments influents de lingüística evolutiva i presenta una implementació bàsica d'un model basat en agents existent per a l'estudi de l'adquisició de sistemes lingüístics sobre construccions lògiques. El model en qüestió ha estat construït utilitzant el llenguatge de programació conegut com a Prolog i dissenyat al voltant d'una sèrie de mòduls reutilitzables. L'objectiu final és que pugui ser emprat per altres investigadors de camps relacionats en futurs estudis que es duguin a terme.

Resumen

Dentro del amplio campo de la lingüística, el origen y la evolución de los lenguajes humanos se mantiene como una cuestión aún sin explicaciones ampliamente aceptadas. Sin embargo, la introducción de modelos basados en agentes durante los últimos años como herramienta para la comprobación de hipótesis, ha propiciado el progreso de dichos estudios.

Este proyecto analiza algunos experimentos influyentes de lingüística evolutiva y propone una implementación básica de un modelo basado en agentes existente para el estudio de la adquisición de sistemas de lenguaje sobre construcciones lógicas. El modelo en cuestión ha sido construido utilizando el lenguaje de programación Prolog y diseñado entorno a una serie de módulos reutilizables. El objetivo final es que pueda ser empleado por otros investigadores de campos relacionados en futuros estudios llevados a cabo.

Acknowledgements

The work done in this project would not have seen the light were it not for the patient and model guidance of my director María Josefina Sierra Santibáñez. Beyond teaching and explaining the academic topics and articles contained herein, she inculcated in me the academic rigour needed to finally consummate this project.

Please accept my deepest appreciation and my most humble gratitude.

Contents

1. Introduction and objectives

During recent years, we have witnessed an increasing amount of interest towards the origin and evolution of human languages. However, we are still lacking widely accepted theories to explain these phenomena and many crucial questions are still unanswered. In this context, agent-based models and simulations have gained importance, as they provide an interesting balance between theorizing and formal mathematical modelling.

This project builds on an existing agent-based model for studying the emergence and evolution of language systems of logical constructions [SSn14]. We attempt to generalise the implementation of this model in order to provide a set of software modules that allow the construction of new agent-based models which can be adapted to study language systems of logical constructions, using different language strategies and potentially other types of language systems.

The present work differs from other agent-based models reported in the literature in the specific type of language system studied, which focuses on the expression of logical constructions, and in the programming language and approach used to implement it, which is Prolog and in general logic programming. In order to achieve the overall aim of this project, we can stablish four different subgoals, which build on the results of the previous ones and are the object of the different sections of this document.

The first subgoal was to identify the main software modules that should be part of our implementation and to understand the functionalities they should provide. This was done by analysing a significant number of agent-based models proposed in the literature to study the emergence and evolution of language systems associated with different aspects of grammar, such as compositionality, verb tense aspect or agreement, and by comparing these systems to the agent-based model proposed in [SSn14].

Once a generic and sufficient set of modules had been designed, the second subgoal was to address the main issues associated with their implementation in Prolog. This required, among other things, being able to synchronise the behaviour of the agents in the population, which interact with each other playing language games, to monitor the overall simulation, and to use appropriate data-structures and algorithms to implement the cognitive abilities needed by the agents to engage in linguistic interactions, learn from each other, and adapt their behaviour to the widespread conventions among the population.

Given that the resulting implementation wants to prove useful to other researchers interested in conducting experiments on the origins and evolution of language, the third subgoal was to specify the reusable Prolog modules around which the implementation was structured as clearly and systematically as possible. This, together with the declarative semantics of Prolog programs, might be especially helpful in a multi-disciplinary area such as evolutionary linguistics.

Finally, the fourth subgoal was to show how complete agent-based models could be configured using the modules implemented in this project with concrete examples that could be used to conduct experiments addressing novel aspects of the emergence and evolution of language systems of logical constructions.

1.1. Scope of the project

As stated at the beginning of this chapter, this project does not attempt to build a new model from scratch (which is beyond the scope of a final degree project), but rather to implement and generalise an existing model. To achieve this, and complete the objectives outlined before, we need access to the work presented in [SSn14] and a simple computer in which to develop and test the model. Because this project aims to become a tool for other researchers, we also need to provide a clear implementation and a detailed documentation that allows an external user to understand and customise the underlying mechanisms of the model.

The software itself is written in Prolog, a choice motivated by the fact that this programming language is familiar to computational linguists, and that it is in general considered to be easier to understand and master by researchers from fields such as cognitive psychology, anthropology or evolutionary biology, who might also be interested in studying the origins and evolution of languages. Other studies, which are discussed chapter 2, use Lisp instead [Kir02, Vog05, BS13, Ste99, GSB12].

Out of the available Prolog implementations on the market we chose to use the Ciao system, owing to the fact that it is free software and supports standard ISO-Prolog, which makes it easy to run the model in any machine, independently of its settings and operating system. Ciao support for concurrency and in particular concurrent predicates, which can be modified by different processes, was another strong point of the system, because each agent in the model is represented by an independent process and communicates with each other via socket connections. Other reasons for choosing Ciao include the integration with Emacs and its straight-forward user interface.

1.2. Working methodology and validation

The model presented in this project has been developed via the accomplishment of intermediate goals, following the planning established (please refer to chapter 7) and using an iterative and incremental developing methodology.

Initially, research on different areas relevant to the project was conducted, including topics such as agent-based models, language systems or the structure and objectives of past successful experiments. Technical aspects of Prolog that seemed useful for the future implementation were also studied.

After this phase, the construction of the model began. A total of eight modules have been implemented, each one engineered to provide a particular functionality. Overall, the implementation process followed an iterative development, with each one of the modules representing a particular task and in the order below presented:

1. Interaction of the population.
2. Definition of agent.
3. Definition of the conceptual system.
4. Population parameters and operations.
5. Sentence generation from given meanings (speaker role).
6. Sentence interpretation (hearer role).
7. Induction operations.
8. Adaptation mechanisms.

To ensure that the development proceeded according to the plan, weekly objectives and meetings were scheduled in agreement with the project director. Each one of those objectives included assignments such as completing the implementation or the documentation for a particular module, implementing and testing certain parts of the model or writing chapters of this document. Naturally, this is in addition to the 3 mandatory milestones mentioned in the university regulations.

The behaviour of the model itself once completed was tested by conducting different experiments on language emergence and transmission (explained in chapter 6). The modular implementation of the agent-based model allowed an easy configuration of different experiments from the basic module system, which only required simple extensions, modifications or substitutions of some modules particularly relevant to the goals of each experiment.

1.3. Actors and stakeholders

The people and collectives below listed and described are the ones directly or indirectly involved in this project.

- *Project developer.* Only one person will assume the role of project developer and its associated responsibilities. This means that I myself am going to work in the planning, information research, implementation, documentation and experimental tasks existing within this project.
- *Project director.* There is a single director whose role is to guide the developer of the project and provide assistance when it encounters difficulties during the overall process. The director of this project is Dr. María Josefina Sierra Santibáñez, associate professor from the Computer Science department at the Technical University of Catalonia (UPC).
- *Project users.* Our aim is to implement and document a system in such a way that it can be used by other researchers in other fields. Therefore, the users of this project are researchers interested in performing simulations to study the origin and evolution of human language systems with a population of independent agents. Nevertheless, it is a requirement for the potential users to have minimum prior knowledge of the Prolog programming language.

1.4. Resources and impact

The work presented in this project does not require an extensive amount of resources, and further financing by external bodies has not been necessary. It suffices to have a computer in which to develop and test the model. A more precise list of the resources used is provided next:

– Computer, with no particular hardware requirements, running the Windows 8.1 and Debian 6.0 operating systems (or equivalent).

– Ciao Prolog version 1.10 (or higher).

– GNU Emacs 23.4 (or another editor of choice).

– TexStudio 2.8.8 LaTeX environment (or equivalent).

– Access to published research papers.

It is not straight-forward to foresee the outcomes and benefits that may originate after the successful completion of this project, mainly because its research-oriented character. Nevertheless, the model implemented here aims to be customisable and useful to researchers in a broad range of fields, from computer science to computational linguistics or cognitive psychology, wanting to conduct computational experiments on the origin and evolution of language systems via simulations with a population composed of autonomous agents.

2. Agent-based models

There are extensive surveys [Ste11b] of the research conducted within the field of language evolution, which can be divided into two different approaches. While the ***biolinguistic approach*** regards biology, genes and the physical traits of the human communicative apparatus as a driving factor in the origins of the linguistic structure, the ***evolutionary linguistic approach*** emphasises the role of cultural forces in which the human population is inevitable immersed. This project is notably focused on the latter evolutionary model, but at all times one should keep in mind that language origins and evolution are likely to be explained by an interaction of ecological, cultural and biological aspects [Ste12].

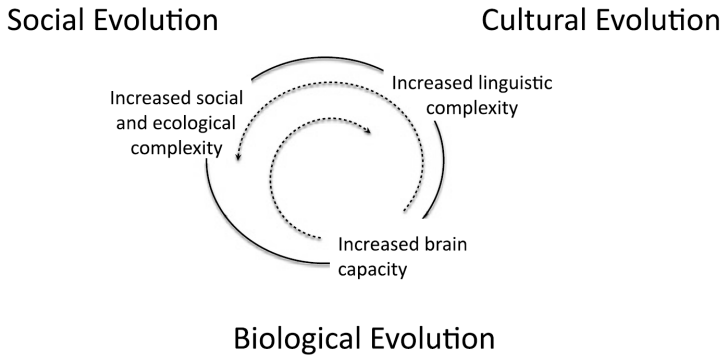


Figure 2.1: The origins and evolution of language is a complex process driven by several biological, social and cultural factors (an image from [Ste11b]).

Several different approaches to biolinguistics exist [DSB11], some even employing agent-based iterated learning and game theoretic models. Their focus is to show that if a population has an innate bias for learning a certain structure, this bias becomes expressed in their language after a number of generations. Those experiments have successfully shown that compositional structure may emerge from teacher-learner chains [Kir02], given that the biological mechanisms favour such a structure. Acknowledging the importance of the such results, we now proceed to give a brief overview of the here more relevant evolutionary linguistic approaches.

Early investigations in the field of evolutionary linguistics explored topics such as typological data and construction of cladistic trees that reflect family relations among languages with common origins [Ste11b]. Tracking and observing the similarities of different linguistic features among various regions strongly supported the idea that cultural evolution is the primal factor that determines the structure of a language.

A second set of investigations focused on how phonetic systems and grammatical categories change over the time, seeking to find plausible explanations to this historical data. However, and co-occurring with significant advances in hardware and software engineering, what really pushed the state-of-art was the introduction of ***agent-based models***, which are extensively used nowadays to construct models that can be used to validate hypothesis about how different particularities of the language appeared or came to be.

Agent-based models of cultural evolution usually involve a population of robotic or non-physical software-based agents who interact pairwise by playing ***language games***, assuming the roles of speaker and hearer respectively [Ste12]. The speaker has a specific communicative goal, conceptualises the world for language, and transforms this conceptualisation into an utterance. The hearer tries to parse that utterance, reconstruct its meaning and map it onto its own perceptual experience of the world. Depending on the outcome of the game, speaker and hearer use different strategies to adapt their internal languages in order to be more successful in future language games.

It is commonly assumed that the agents in these models are initially endowed with an specialised learning mechanism which can be compared to human cognitive abilities. This is necessary for observing the emergence of possible language systems that allow the agents to be successful in a language game. Some examples are the ability to construct complex concepts, or to use and detect linguistic devices such as word order, syntactic categories or case markers. The experiments conducted consist of various simulations, in each of which the agents in the population play a series of language games, configure possible strategies, try them out and select those that are more useful.

The goal of the experiments is to find out whether the population as a whole communicates effectively, and to observe the conceptualisations and linguistic constructions that emerge in the population as a result of the processes of collective invention and negotiation, as well as the evolution over time of various macroscopic features of these language systems, such as the average size of the agents' grammars or the similarity of their grammatical constructions [SSn15].

2.1. Language systems

Theories of language evolution study language change at two different levels: that of language systems and that of language strategies. ***Language systems*** capture the regularity observed in some part of the vocabulary or grammar of a language, for example, a system of basic colour terms, tense-aspect distinctions or cases. In short, they group a set of paradigmatic choices both on the side of meaning (the conceptual system) and on the side of form (the linguistic system).

The ***conceptual system*** includes pragmatic and semantic distinctions that are expressible in a language system and can therefore be used as building blocks for conceptualisation. The ***linguistic system*** includes all the syntactic and morphological categories and grammatical constructions needed to turn a conceptualisation into a concrete utterance [Ste11b].

A given language, such as Catalan, English or Spanish, comprises many language systems, which are closely integrated. Linguists call the approach underlying a language system a ***language strategy***. They refer to relative-clause formation strategies or coordination strategies for combining nouns. Agent-based experiments aim to provide explanations of the processes by which concrete language systems may emerge and evolve using a particular language strategy.

2.2. Examples in the literature

We summarise below some studies of different types of language systems that are described in the origins and evolution of language literature and use agent-based models as a tool. This serves the purpose of familiarising with the different techniques used by researchers, but also to introduce concepts which will relate later on to the model developed in this project.

2.2.1. Compositional structure

Compositionality and recursion are two of the most distinctive features present in human-spoken languages [Kir02]. A particular expression is compositional when its meaning can be defined in terms of the meaning of its parts and how they are glued together. Expressions where this does not hold are referred to as ***holistic expressions***. On the other hand, recursion occurs when a certain part of an expression can contain a constituent of the same category.

The emergence of compositional structures has been investigated using the observational game, the guessing game and a particular type of agent-based models known as ***iterated learning models***. In this type of agent-based models some agents assume the role of teachers (speakers) and others the role of learners (hearers), and simulations can be carried out using a simple population of two agents (with a single teacher and a learner) or more.

One example of an iterated learning model can be found in [Kir02]. Kirby shows that the compositionality and recursion properties emerge over the time with social transmission as the sole trigger for the process. To understand his experiments, we need to define two concepts:

- ***I-language***: The knowledge of a language possessed by each user of that language.
- ***E-language***: The language represented in the utterances produced by the population of speakers of a language.

The model proposed by Kirby consists of only two agents who play language games. One of the agents takes always the role of teacher and the other one the role of hearer. The teacher is first given a meaning to express, which it attempts to transform into an utterance by means of its I-language. This utterance is later used as an input for the learner, who tries to understand it. After repeating this process a certain number of times generation shift occurs: the learner becomes the new teacher, the old teacher is discarded and a new learner is created with an empty knowledge of the language.

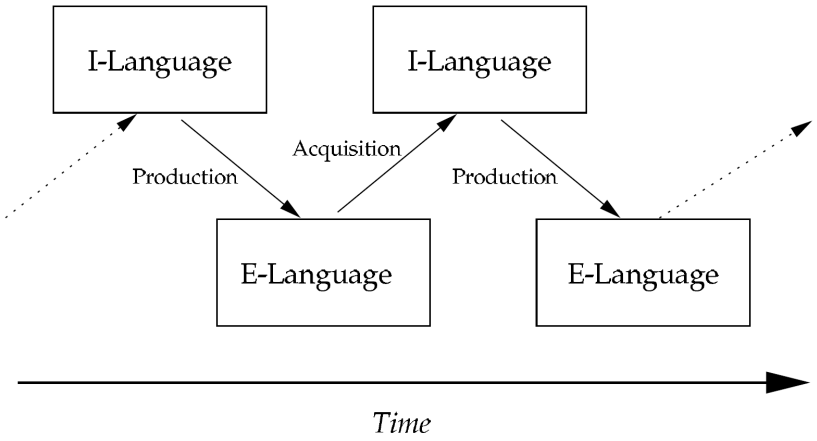


Figure 2.2: Transmission of language over time in a teacher-learner chain (an image from [Kir02]).

Utterances in this model are pairs consisting of a meaning and a string of characters representing it. To construct those meanings, the agents use a series of atoms representing nouns or verbs and combine them to form predicates which take a number of arguments and can have hierarchical structure. The complete range of available atoms is known and shared by all agents before the first language game is played. Some illustrative examples can be found next:

- Atoms: *cat*, *dog*, *food*, *eats*, *fight*, *fears*.
- Predicates: *eats(dog, food)*, *fears(cat, fight(cat, dog))*.
- Utterances: *['dogeatsfood', eats(dog, food)]*.

Each agent uses an initially-empty context-free grammar which allows them to represent their knowledge of the language. Once an utterance is heard by an agent and not understood, the agent incorporates the simplest possible rule that generates the utterance, which is a holistic rule. Taking *['dogeatsfood', eats(dog,food)]* as an example of an utterance, this would produce a rule of the form $S/eats(dog, food) \rightarrow dogeatsfood$.

Rule induction is applied then to the agents' grammars. This is a process which aims to obtain higher-level rules by looking for ways to generate similar expressions and generalising the associated rules. For example, given the two rules: $S/eats(dog, food) \rightarrow dogeatsfood$ and $S/eats(cat, food) \rightarrow cateatsfood$, we can generalize the concepts *cat* and *dog* by doing $S/eats(x, food) \rightarrow N/x\ eatsfood$. Note that now a new syntactic category N must appear in order for the two resulting grammars to be equivalent, and also two more rules of the form $N/cat \rightarrow cat$ and $N/dog \rightarrow dog$ need to be added.

Let us review now the simulation cycle in [Kir02]:

1. The speaker is given a meaning to express, and tries to communicate it using its knowledge of the language. Should this not be possible, it invents an expression taking advantage of rules which can be used to express parts of the meaning.
2. The learner tries to parse the utterance. If this is not possible, then it adds the minimal rule necessary to parse the utterance and applies rule induction.
3. Steps 1 and 2 are repeated a number of times inferior to the cardinality of the meaning space (the reason for this is discussed later). Next, the teacher is deleted, the learner becomes the new teacher and a new agent with no knowledge of the language becomes the new learner.

By using this model, Kirby is able to demonstrate that as the simulation goes on, the grammars of the agents decrease in size and allow them to express a higher number of meanings. In fact, those grammars are able to evolve into a complex compositional syntax where nominal and verbal categories appear, together with different sentence orders such as (Subject - Verb - Object) or (Object - Verb - Subject). If the meaning space is infinite, that is predicates can take other predicates as arguments, then recursion also appears in the obtained rules, representing subordinate sentences.

Kirby argues that this evolution occurs because more general rules allow the expression of a wider range of meanings and thus, these rules have a higher change of appearing in the E-language from the I-language, constituting a better replicator (this also allows convergence towards shorter grammars, because the generality of the rules in them is higher). Finally, it is argued that holistic languages cannot survive because the E-language used by hearers to learn only allows the observation of part of the speaker's language. This occurs due to the number of meanings selected before changing the agents being always less than the count of all possible meanings. The difference between the set of all possible meanings and the subset of meanings selected for communication is commonly referred in the literature as a ***transmission bottleneck*** and pointed as the main factor that pressures the appearance of compositional languages.

Another study similar to the previous is conducted in [Vog05]. It also studies the emergence of compositional structures, but with a few notable differences. First, it assumes that syntactic structures co-evolve with semantic ones, instead of being given from the start. More precisely, the following is stated:

- The emergence of compositional linguistic structures is constrained by semantic structures and based on exploiting regularities in expressions.
- The emergence of combinatorial semantic structures and how words are modified by others is constrained by compositional structures and based on the regularities shown while interacting.

The former model has a strong basis on the Talking Heads experiment [Ste99]. In the Talking Heads model, a group of agents observe a scene composed of a blackboard which contains plain figures of different colours and shapes. The goal of the population as a whole it to evolve a lexicon to refer to them individually.

Objects can have four features in Vogt's model: level of red, green and blue (rgb colouring) and shape. These four features are the basic building blocks for constructing meanings (also known as categories). In turn, categories are represented as points in a high-dimensional space, where each dimension accounts for a single feature. For simplicity, meanings need to cover all dimensions, either as a holistic space or as a composition of non-overlapping spaces.

Each agent maintains its own ontology and uses it to produce and parse utterances. Such an ontology contains a set of categorical features, represented as points along one particular dimension in the feature space. The agents use it for extracting features and combine them into categories. This process is known as a ***discrimination game***, and succeeds if the category obtained for a given object is distinctive and fails otherwise.

Similarly to the previous model by Kirby, the agents also maintain their own grammar, which allows them to construct utterances for a particular category and to parse them. The terminal slots of these grammars however, represent points in the feature space. For example, the rule $N/rgb \rightarrow blue/[0, 0, 1]$ means that a terminal slot in the rgb space will be uttered as the string "blue" if its meaning is $[0, 0, 1]$ (no red nor green components in the rgb space). With this example, it also becomes clear than the meaning of a rule can be formed from the categorical features listed in square brackets.

Additionally, each rule has an inherent weight value which determines how effective it has been in past language games. Categorical features also have a weight value associated to them. If a rule contains terminals, its score is computed as the product of its own weight and the average weight among terminals. This score will become valuable later on as a tool allowing the population to agree on the same structures for meanings, a process known as ***self-organisation***.

Given the communicative context, we now discuss the particular type of linguistic interactions used in this model, which are known as the observational game and the guessing game:

1. The speaker is given a a certain object to talk about, a ***topic***, and then extracts the distinctive features that describe the object properties.
2. The speaker non-verbally informs the hearer about the topic selected in the observational game. In the guessing game no information is given.
3. Both agents form a category that distinguishes the topic from the rest of objects in the context. If this category can not be obtained using the agents' ontologies, a new category is created using discrimination games (in the guessing game all objects in the context are considered as potential topics).
4. The speaker produces an utterance, obtained using the combination of grammar rules with highest score that match the chosen meaning.
5. The hearer tries to decode the utterance produced by the speaker. To do so, it creates a set of rule compositions that decode the utterance, and then extracts the ones not matching the topic (observational game) or the ones not matching any of the objects existing in the context (guessing game). Finally, the hearer generates a meaning which allows to discriminate the topic by selecting the composition of rules with highest score. If this composition can not be found, the hearer fails.

Note that in Vogt's model, two different types of language games are considered. In the ***observational game***, the speaker communicates its topic to the speaker, and thus both share joint attention to a particular object in the scene. However, in the ***guessing game***, this communication is not present, and the hearer has to guess which object the speaker is trying to refer to. It is only after this process that the hearer receives feedback on whether the guessed topic was right or not.

A language game may fail when the speaker is unable to produce an utterance or the hearer fails to interpret the utterance. In the first one of the situations, the speaker needs to invent new rules using similarities with existing rules in the same way as previously seen in the Kirby's model. When the hearer is not able to decode an utterance, it starts a process known as ***induction***, which allows it to derive new grammatical structures. Hearers can operate using the following:

- ***Incorporation***: add a holistic rule.
- ***Exploitation***: take advantage of an existing rule capable of decoding part of the sentence and add a new rule covering the remaining parts.

- ***Chunking***: split a holistic expression and create new rules in the grammar, based on the largest common chunk and provided the utterance is not parseable, but aligned with some previous holistic rules.

Along language games, there is an adaptation process which increases the weight of the rules used and decreases the weight of competing ones in both the speaker and hearer. More formally refereed to as ***lateral inhibition***, this serves as a competitive selection mechanism and allows agents to unlearn certain rules and to self-organise, an issue not addressed in the previous model by Kirby. Rule generalisation and merging are present as explained in Kirby's model.

Experimental results confirm the previous findings claiming that the transmission bottleneck triggers the emergence of compositional rules. Nevertheless, Vogt is also able to create compositional rules even without introducing such a bottleneck, although the proportion of compositional expressions is fairly unstable, and can eventually collapse in a holistic language. Compositionally emerges in this case because of the high statistically recurring structures in both utterances and meanings.

The guessing game seems to be more stable with different populations sizes, and this is expected, because there is more pressure to disambiguate the language, as the hearer does not know the original intended meaning of the speaker and has to guess from the utterance. The size of the bottleneck also plays an important role, being smaller bottlenecks a more fertile ground for compositionality.

2.2.2. Agreement systems

Grammatical agreement occurs when features associated with one linguistic unit, such as number or gender, become associated with another unit. Typically this association is made explicit by using some kind of morphological marker which shows the dependencies between words. Beuls and Steels [BS13] present an agent-based model and a series of experiments that show how agreement systems could arise, and which cognitive and cultural traits intervene in the overall process. It is hypothesised in their work that agreement systems are motivated by the need to minimise combinatorial search and semantic ambiguity when parsing.

As we have seen in previous examples, the Beuls-Steels model uses a set of objects present in the situation with various properties that are observable and known by the agents. Each object has a number of features that are expressed using predicates such as $green(o_i)$, meaning the object o_i is green. A population of 10 agents is initialised with a predefined vocabulary, in the form of associations between a set of properties and a word. For instance, the word "jubope" could mean $green(x)$ and $small(x)$, where x can be bound to an object.

Apart from vocabulary, each agent has also a grammar which allows them to construct and parse expressions. Both vocabulary and grammar are implemented using Fluid Construction Grammar, which is not explained in detail here. We encourage the reader to look up the technical details appearing in [BS13].

Agents in the Beuls-Steels model play what is called game of reference, also known as ***naming game***. In this particular language type of linguistic interactions, the topic can be composed of either one or more objects. The overall process is as follows:

1. The speaker selects a set of objects as a topic to communicate.
2. The speaker looks for distinctive properties for each object in the topic. Note that each object is of a certain type, which defines the possible attributes and values of its instances.
3. The speaker retrieves the set of words to express these properties. Words can cover more than one property, but all properties must be true for the objects associated with the word.
4. The speaker utters the words in random order.
5. The hearer tries to reconstruct the set of properties from the words uttered by the speaker.
6. The hearer identifies which objects satisfy these properties, and succeeds if this set of objects is the same as the one chosen by the speaker.
7. The game fails if the interpretation is ambiguous or the set of inferred objects differs.

This language game can be viewed as a simple interaction, but in fact, the hearer does not know the number of objects the speaker is referring to or which words refer to which object or objects. Thus the hearer must try all possible combinations of objects, which are exponential with respect to the number of words.

While this ambiguity and exponentiation indeed exists in real human interactions, we can use some strategies to help reduce it. We humans know as a matter of fact that some attributes are not possible for certain objects (the sky can be blue, but this is not a natural property to describe an idea). Selection restrictions are implemented in the agents by giving them access to an ontology defining the different types of objects and their possible attributes and values.

The other mechanism used to reduce ambiguity is the application of constraints that arise from the situation model. That is, because the speaker is referring to a topic in the situation, we can assume that a certain attribute must be linked to an object in the situation which has the same value for that attribute. This is used for pruning some hypothesis in the model.

Nevertheless, in order to reduce the hypothesis space drastically, some kind of agreement system is needed. This is initially presented in the form of markers, which are associated to uttered words that refer to the same object in the topic.

The first language strategy implemented in [BS13] attempts to simulate the rise of marker systems. To do this the speaker tries to detect the difficulties that the hearer may encounter when parsing the words by trying to the parsing job itself, a process known as ***re-entering***. If the speaker notices combinatorial complexity or some ambiguities, it adds a marker to each word (syntactically represented as a suffix) that introduces properties of the same object. Markers are stored in private inventories that the agents possess and are used in subsequent games, even if not needed. The hearer undergoes a similar process than the speaker, uses markers in subsequent communications and is now able to instantly identify which words refer to the same objects, even if it is the first time seeing a marker invented by the speaker.

Regardless, there is still an issue because agents need to agree globally on markers, through a process of self-organisation. This is done with using lateral inhibition, which we have already seen in Vogt's model from the previous section. Weights are set for each marker and when a marker is used, its weight increases while decreasing the weights of competing markers. When choosing which marker to use, the speaker prefers the marker that has the highest score and has not yet used in the current utterance.

The model at this point is able to show that marker systems indeed emerge and get transmitted culturally. Occasionally some new markers appear, but get damped because of lateral inhibition. Still in real human languages, we tend to prefer markers which are not formal, but carry some kind of meaning or connotation. To simulate, Beuls and Steels propose a second language strategy in which an inventory of associations between a feature matrix conveying semantic information, a marker and a score is kept by each agent.

When the speaker decides to use a marker for a word referring to a particular object, it first searches the inventory fo find one not used in the same utterance, and whose feature matrix fits the feature matrix of the word used to refer to the object. In cases where no such marker is found, the speaker tries to find an attribute which is distinctive for words used to refer to the different objects in the utterance and creates new markers for each distinctive attribute-value pair. On the other side of the coin, whenever the hearer encounters new markers, it treats them as formal, and constructs and adds a feature matrix by using the same induction process as the speaker.

This strategy leads to a self-organisation where the agents share which markers they prefer but also the feature matrices associated to them. Markers that express a single feature dominate, because they can be applied to a wider range of situations. However, more modifications to the model are still possible in order to resemble more the strategies used by humans to create agreement systems.

Linguists have observed that markers in real life are derived from existing words such as pronouns or classifiers. This is actually preferred by humans because it is less ambiguous and does not require uncertain guesses when new markers are encountered. Rather than inventing a random string when a new marker is needed, the speaker will now take an existing word that expresses one or several features of the topic and use that as a marker. Again, an agreement system self-organises, but this time with even fewer variations between markers, which are more quickly discarded. Once a marker system efficient enough takes off, it is hardly challenged.

Another observation made by linguists is that markers erode. This happens because of an increase of articulatory efficiency, and because parsing takes less effort if we use a new shorter form of the word rather than the original one associated to the marker, which is most likely to appear in other situations. This process is simulated by adding a tendency for optimizing on top of the aforementioned reuse strategy. The word used is simplified with a certain probability, and parsing in the hearer becomes now flexible enough so as to recognise such variations as the original word and still remain coherent.

This mechanism leads to a problem: the alteration cannot become a norm because of the lateral inhibition process, which quickly discards it. In order to correct this, the new variant of the word and the original word are stored, and both words are used randomly when selecting a marker. If the marker is encountered a second time, then it becomes the norm and the older word is discarded for this agent. Experiments performed show that forms become continuously reduced.

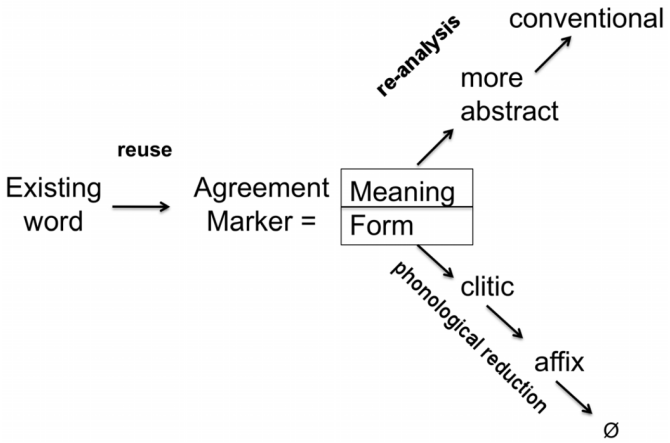


Figure 2.3: Humans build agreement systems by reusing existing words. The form of those words erodes and their meaning becomes more abstract, with purely conventional semantic features (an image from [BS13]).

Finally, the authors address the phenomenon of coercion. Markers in real life are applied to situations which do not fit the original meaning and features become conventional. This is what happens with gender in most languages, being arbitrarily assigned to inanimate objects while originally meant for male/female distinction. The explanation is that having a large number of markers makes it difficult to learn. Us humans take the ones which are more general (semantic bleaching) and recategorize a word to assign more agreement features, even though the previous word did not have this feature.

To include coercion in the model, the differentiation between controller and target words is introduced. Controllers determine the agreement features of targets. In the Beuls-Steels implementation, some words at the beginning of the simulation are initialised as controllers. Then an extra step is added to the invention of a marker by the speaker, who will now assign by convention the values of a partial marker to the controller if the controller is undecided on those features. This automatically makes the controller compatible with the marker.

Lateral inhibition is also used in this case as a way to self-organise the population, storing a score for each feature matrix of each controller, and increasing it when used and decreasing it when not. Additionally, agents are now initialised with a minimum marker system as well for convenience. By running this experiment, a reduction in the number of markers is observed. It shows that some feature matrices denoting a certain meaning become dominant over others associated with the same marker.

This research as a whole is, according to their authors, another example which supports the idea that the structure of language is a consequence of its function and usage in communication, operating within the processing constraints of human cognition and physical characteristics.

2.2.3. Verb tense aspect

Another study by Gerasymova et al. [GSB12] deals with the parsing, production and learning of an aspect system. To do so, it employs an existing system of aspect markers found in the Russian language as a model.

Consider the sentences “Pieter wrote a thesis” and “Pieter has been writing a thesis”. It is true that both of them express the action of writing. However, while in the first sentence the action was completed in the past (perfective aspect), in the second one the process of writing is still ongoing (imperfective aspect). Aspectual distinctions such as this one allow speakers to highlight the whole or a specific part of an event and thus increase their expressiveness and communicative success.

In English, most fine-grained aspectual distinctions are optional and expressed periphrastically (e.g. “she swims regularly”, “he finished writing”) or by means of auxiliaries (e.g. “he has written the letter”). For Russian speakers though, it is mandatory to denote aspect by using prefixes attached to the verb. In this section we refer to aspect as a grammatical category denoting the distinction between the imperfective and perfective character of the verbs. This is closely related to the notion of Aktionsart, which is the lexical aspect and refers to the part of the verb that is structured. The category of Aktionsart introduces a more refined aspectual categorization of events, including aspects like activity, achievement, state, etc.

Russian overtly marks perfective aspect by placing prefixes next to the verb, while imperfective aspect is assumed by default if no prefix is found. The model presented by Gerasymova et al. attempts to reconstruct such an aspect system by using an agent-based model. Similarly to the previous examples of agent-based models we have seen, the agents interact with each other by using language game, in this case denoted as ***aspect language games***, and executed as follows:

1. Both agents perceive a shared context, which models their joint attention frame. The context is composed of two or more events of the same kind but with different temporal semantics, and each event has a different protagonist. We can picture for example, as suggested in the paper, “Micheal reading for a while” and “Masha reading the whole time”. Those events are coded in types, an enclosed in a time frame so that they are perceived and represented appropriately.

2. The speaker chooses one event, the topic to communicate. It then asks a question about the protagonist of that event by using the information of the event in which the protagonist is involved. Continuing with the previous example, this could be something like “Who was reading for a while?”. Note that a temporal structure needs to be incorporated into the question in order not to be ambiguous.

3. The speaker conceptualises the meaning and utters an expression.

4. The hearer parses and interprets the utterance by comparing the result of interpretation to the context. The task of the hearer is to identify the protagonist of the focus event unambiguously (guessing is not allowed in this model).

5. If the hearer is able to discern the answer, it verbalises it and forwards it to the speaker. Otherwise it fails.

6. The speaker signals whether the correct answer was given or not (in case received). The game succeeds only if this feedback is positive.

7. It the hearer could not identify an answer (or the answer was wrong), then the speaker reveals the right answer.

The agents conceptualisations are implemented by means of Incremental Recruitment Language (IRL) networks. IRL is a formalism that allows representing and computing semantic structures that can be used for communication. These networks link mental operations with concrete semantic entities, such as the temporal characteristics denoted by Aktionsarten. Grammatical processing of the utterances is achieved with Fluid Contruction Grammars, which allow to map lexical stems to particular event types, recategorise the verbs in terms of semantic categories, map semantic structures to a abstract syntactic ones and finally to a certain marker based on the aspect and Aktionsart.

In the model described up until now, the authors introduce different types of agents: tutors and learners. Tutors know everything about the aspect system, and learners miss all grammatical aspects related to aspect and Aktionsarten. By playing language games among them, learners try to produce questions but also try to answer the ones inquired by tutors and other learners. Provided that a good learning strategy exists, at the end of the simulations agents converge to the same set of grammatical constructions. The learning strategy that makes it possible to achieve this is detailed next.

Agents learn by enacting ***diagnosis*** and ***repairs***. Diagnosis is a process that monitors the production and parsing of sentences and signals potential problems, in particular, inability to parse an utterance or ambiguous interpretation. The repairing mechanism attempts to extend the linguistic repertory to avoid such problems.

Ambiguous interpretation is identified by the speaker with the help of the lexicon by re-entering the expression he produced and trying to parse it. If two hypothesis seem plausible (no disambiguation) then there is an error. Inability to parse is detected by the hearer, who will fail the language game consequently. Learners in the model use 3 repairing strategies to solve this problem, which try to simulate the learning process of a language in children and are organised in different layers of abstraction:

- ***Holophrases***: They are single words that communicate intentions about a scene. When the hearer cannot parse a prefix, it searches for semantic distinction between possible topics. Then the hearer, after the speaker reveals its intentions, stores the complete utterance as a chunk or holistic expression in its grammar, which maps meaning and form. Note that a system with only holophrases reaches maximum coherence after all possible prefix-verb combinations are seen.

- ***Item-based construction***. These are general constructions that the learning agent tries to create based on repeatedly encountering the same structure. By encountering the same pattern multiple times and keep storing them in a particular inventory, the agent is able to infer the usage pattern prefix+verb for a particular meaning, which is then learnt. Even with this and holophrase mechanisms, learning agents still can not derive new Aktionsarten.
- ***Abstract contructions***. When the speaker detects ambiguity of this own utterance upon the re-entering process, it looks at its inventory of item-based constructions. This makes it realise that temporal semantics of a verb are usually expressed by means of prefixation and then the speaker learns an abstract construction, such as "A verb becomes marked for perfective aspect". Therefore the agent can generate a unit for prefix, but without linguistic material, simply by mimicking what it observes from teacher agents. In fact, they are now able to generate perfective derivation of any known verb.

Simulations conducted with this system show that agents are able to acquire the aspectual grammar. In the beginning only holophrases exist, but as the agents start noticing the similarities between stored constructions and start to generalise other grammatical constructions are created, namely item-based constructions and abstract constructions. Communicative success then converges rapidly. It is important to note that all constructions in the agent inventory have a score, which is updated depending on the success of the communication (unsuccessful constructions are punished). This is what triggers the dwindling of holophrases in favour of more general rules.

The final modification introduced in the Gerasymova et al. model attempts to tackle the emergence of the Aktionsarten system. Now no tutors are endowed with the aspect system, and agents are only initialised with basic lexicon. The repair procedure must undergo some modifications in this case. When the speaking agents are unable to express a non-ambiguous question, they invent a new random marker covering that meaning. In turn, when hearers can not parse, they use an item-based construction process.

Experiments show convergence towards the optimal number of markers for a given number of actions and aspects. However, some synonyms may appear during the process. A score mechanism of lateral inhibition is used as a way to self-organise the agents grammars, rewarding the score of markers used successfully and punishing competitors.

3. A model to study logical constructions

This chapter provides extensive details about the agent-based model implemented in this project. We will initially focus on how the logical expressions we want to study are structured, and then, on the particular type of language game used and on the working principles of certain behavioural aspects of the agents.

3.1. A language system for expressing logical constructions

Our current work is based in the agent-based model for the study of the acquisition of a language system of logical constructions presented in [SSn14]. This language system has its own ***conceptual system*** and a ***linguistic system***, both of which are discussed next.

3.1.1. Conceptual system

In our simulation environment, we assume that a group of several agents are trying to communicate logical combinations of basic categories[1], which are true for a particular subset of the set of all the objects in a given context. This context is similar to the one used in "The Talking Heads Experiment" [Ste99], which consists of the set of objects pasted in a whiteboard situated in front of the agents. We also assume that at the beginning of a simulation run, the agents have developed a common vocabulary that allows them to refer to individual object features (basic categories). These features are represented by propositional symbols in the agents' memories. Some mappings of propositional symbols and the propositions they denote could be:

- `up` → Every object situated in the upper part of the blackboard.
- `da` → Every dark object which is pasted in the blackboard.

At the same time, we suppose that the agents have developed certain logical categories which give them the ability to construct propositional logical formulas from propositional symbols. However, it is important to note that the agents do not yet know how to express these logical categories nor the logical formulas they can construct with them through their shared language. Thus, linguistic communication of logical formulas among agents is non-viable at the beginning of a simulation, and it is developed afterwards.

[1] Examples of basic categories are `left`, `right`, `up`, `down`, `light` and `dark`.

Our experiments aim to show that a population comprised of autonomous agents can construct a shared language, that is a vocabulary and a set of grammatical constructions that allows them to communicate such complex meanings. These formulas are non-recursive and so they are formed with a single boolean function and one or two propositions.

The particular set of logical categories the agents can use is the set of unary and binary boolean functions. Their meaning can be expressed using the standard connectives of propositional logic as shown in Table 1. We will use prefix notation to represent these formulas. For example, the formula $\texttt{up} \vee \texttt{ri}$, which means "The set of objects which are in the upper or the rightmost part", will be written as `[or,up,ri]` from this point on.

Boolean function	**Logical meaning**
`not`	$\neg A$
`and`	$A \wedge B$
`nand`	$\neg(A \wedge B)$
`or`	$A \vee B$
`nor`	$\neg(A \vee B)$
`if`	$A \rightarrow B$
`nif`	$\neg(A \rightarrow B)$
`oif`	$B \rightarrow A$
`noif`	$\neg(B \rightarrow A)$
`iff`	$A \leftrightarrow B$
`xor`	$(A \vee B) \wedge \neg(A \wedge B)$

Table 1: Unary and binary boolean functions and their meanings.

3.1.2. Linguistic system

To represent the grammars generated by the agents during the simulation we will use Prolog Grammar Rules [CKPR72, CM03, SSn15]. The head of such rules is an atomic formula whose predicate symbol denotes a syntactic category (e.g. `'s'` for compound sentence or `'p'` for simple sentence). In this project the head uses two arguments, the first one carrying semantic information and the second one being a score which estimates the usefulness of that rule in previous communications. Semantic information can either be a proposition, a boolean function, or a non-recursive logical formula constructed from the former.

For example, if we want to express the formula `[or,up,left]` we could use a single rule of the form $s([\texttt{or}, \texttt{up}, \texttt{left}], S) \rightarrow$ arroizq, where the right-hand side indicates the expression "arroizq" is associated to the formula. However, we also may like to express the same meaning using the following compositional grammar:

$$p(\texttt{up}, S) \rightarrow \text{arr}, \{S\ is\ 0.70\} \quad (1)$$

$$p(\texttt{left}, S) \rightarrow \text{izq}, \{S\ is\ 0.25\} \quad (2)$$

$$c2(\texttt{or}, S) \rightarrow \text{o}, \{S\ is\ 0.50\} \quad (3)$$

$$s([P, Q, R], S) \rightarrow\ p(Q, S2), c2(P, S1), p(R, S3), \{S\ is\ S1{\cdot}S2{\cdot}S3{\cdot}0.1\} \quad (4)$$

In the previous example, the syntactic category `'p'` is used to characterise ***simple sentences*** such as "arr" or "izq" and the syntactic category `'s'` denotes ***compound sentences*** such as "arroizq". Finally, the syntactic category `'c2'` is used to outline words expressing binary boolean functions which are placed in the second position of the sentence, and which are used in grammatical constructions that do not invert the order of the expressions associated with the arguments of the boolean function in the sentence with respect to the order of these arguments in the formula.

Agents can construct grammar rules that concatenate the expressions associated with the components of a formula in different orders. In particular, the word used for expressing a boolean function can be placed in the first, second or third position of the sentence, and the expressions associated with the arguments of the boolean function can also be placed in the sentence in same order as in the formula or in reverse order. For example, grammar rule 5 puts the word associated with the boolean function in the third position of the sentence and it inverts the order of the expressions associated with the arguments of the boolean function in the sentence. This rule generates the sentence "izqarro" to express formula `[or,up,left]`, instead of the sentence "arroizq" as grammar rule 4 does.

$$s([P, Q, R], S) \rightarrow\ p(R, S2), p(Q, S3), c3(P, S1), \{S\ is\ S1{\cdot}S2{\cdot}S3{\cdot}0.1\} \quad (5)$$

3.2. Language game

In the agent-based model described in [SSn14], which constitutes the starting point of the present project, language emergence is seen as a joint activity by which a group of agents construct a common language system as a result of a process of self-organisation of their linguistic interactions. The agents in the population interact with each other playing language games. A language game is a linguistic interaction which typically takes place between two agents randomly chosen from the population, acting as speaker and hearer.

The particular type of language game used in this model consists of the following steps. First of all, the speaker chooses a boolean formula referring to a subset of the objects present in the context of the language game. After this, it generates or invents a sentence to express this formula, and then utters that sentence for the hearer to interpret. The hearer tries to parse the sentence the speaker communicated using its lexicon and grammar.

The game succeeds if the hearer can parse the sentence uttered by the speaker and the meaning interpreted by the hearer is logically equivalent to the meaning the speaker had in mind. Otherwise, the language game fails, and in such a case the speaker communicates the boolean formula it had in mind to the hearer, so that the hearer can adopt an association between the formula and the sentence used by the speaker.

At this moment it might be useful to clarify which aspects of the language system of logical constructions studied by the agent-based model proposed in [SSn14] are present in the agents at the beginning of the simulation, which elements of that language system they should construct during simulations and which cognitive abilities allow them to do so.

The agent-base model described in [SSn14] assumes the agents initially have a common vocabulary for basic categories and the ability to construct complex meanings combining one or two basic categories with a single boolean function. It also supposes that the agents initially do not have a vocabulary for boolean functions nor grammatical constructions. They should construct these elements of their linguistic system during the simulations and do it in a coordinated manner. To achieve this, they can employ some general purpose cognitive abilities for invention, adoption, induction and adaptation. In the following section, we explain briefly how these abilities are used in during the simulations.

3.3. Cognitive capabilities

3.3.1. Generation and invention

At the beginning of a simulation run the agents cannot use their internal grammars to express most meanings, because they initially do not have a vocabulary for boolean functions nor grammatical constructions. In order to form a common language, agents are allowed to invent new sentences for those meanings they cannot express using their grammars.

Invention is performed as follows. If a given formula is atomic, then invention is not necessary (there is already an existing word in the lexicon which uniquely expresses it). In the remaining situations, where the formula consists of a boolean function followed by one or two arguments, a sequence of random letters from the alphabet is chosen to invent a word for the boolean function. A word is generated for each propositional symbol and the two or three words, depending on the type of formula, are concatenated in random order.

As the agents play language games, they learn associations between expressions and meanings, and include those in their individual grammars in the form of rules. Eventually, an agent will be able to express a meaning using only its grammar. In this case, no invention is needed, and the agents generate the sentence with highest score out of the ones they can form to express that meaning. The score of a sentence is computed combining the scores of the grammar rules used in its generation, using the arithmetic expressions on the right hand side of these rules [Vog05].

As an example, we will consider the generation of the sentence "arroizq" for expressing the meaning `[or,up,left]` using the rules 1 to 4 from the previous section. The score of the sentence is computed by multiplying the scores of each one of the compositional grammar rules used to generate this sentence and the score of the grammatical rule itself. Therefore the score of "arroizq" is $0.7 \cdot 0.5 \cdot 0.25 \cdot 0.1$.

3.3.2. Interpretation and adoption

In the second step of a language game, the hearer tries to interpret the sentence communicated by the speaker using its own grammar. Again, at an early stage of a simulation run the hearers will not be able to parse most of the sentences communicated by the speakers, because of the lack of grammatical constructions. When this happens, the speaker communicates the formula F it had in mind to the hearer, and the latter adopts an association between this formula and the sentence E used by the speaker, adding a rule of the form $s(F, S) \longrightarrow E, \{S\ is\ 0.1\}$ to its grammar, where 0.1 is the initial score of the grammar rules created by the agents.

Once the agents can parse a sentence using their own grammar, they select the meaning with the highest score from the set of all the meanings they can obtain for that sentence. Sometimes it will occur that the grammars of speaker and hearer are not compatible, and the interpretation the hearer produces is far away from what the speaker intended to communicate.

The strategy used to coordinate the grammars of both agents in this situation is to decrease the scores of the grammar rules used by the hearer to obtain its interpretation of the sentence, striving for an agreement between all agents, who play in different pairs each turn; and to communicate the meaning the speaker had in mind, so that the hearer can adopt an association between the meaning and the sentence used by the speaker.

3.3.3. Induction

Invention and adoption allow the agents to construct and learn associations between sentences and meanings. However, the agents are also able to extract generalisations from the rules they have learnt so far and induce grammatical constructions and lexical entries that can be incorporated to their grammars and used in subsequent language games to generate and interpret other sentences. Induction is applied whenever the agents invent or adopt an association between a sentence and a meaning, to avoid redundancy and increase generality in their grammars.

Two distinct induction processes can be found in [Kir02], called ***chunk*** and ***simplification***. The induction mechanisms used in this project are based only on the latter, but adapted to grammar rules containing scores as a mechanism of self-organisation. As outlined in [SS07, SSn14], the following definition holds:

Assume a pair of grammar rules r_1 and r_2 such that the semantic argument on the left-hand side of r_1 comprises a subterm m_1, r_2 is of the form $n(m_1, S) \rightarrow e_1, \{S \text{ is } C_1\}$, and e_1 is a substring of the terminals of r_1. Simplification can be applied to r_1 and a simplified rule is obtained. This new rule is identical to r_1 except for the facts that:

1. *The subterm m_1 is replaced with a new variable X in the semantic argument on the left-hand side.*
2. *The substring e_1 is replaced with $n(X, S)$ on the right-hand side.*
3. *The arithmetic expression $\{R \text{ is } E \cdot C_2\}$ on the right-hand side is replaced with a new arithmetic expression of the form $\{R \text{ is } E \cdot S \cdot 0.1\}$, where C_1 and C_2 are constants in the range [0,1], and E is the product of the score variables that appeared on the right hand side of r_1.*

The process of simplification can perhaps become clearer with an example. Assume an agent's grammar containing the rules 6 and 7. Suppose now that this agent plays a language game with another agent and invents rule 8.

$$p(\texttt{light}, S) \rightarrow \text{claro}, \{S\ is\ 0.50\} \quad (6)$$

$$p(\texttt{right}, S) \rightarrow \text{derecha}, \{S\ is\ 0.30\} \quad (7)$$

$$s([\texttt{and}, \texttt{light}, \texttt{right}], S) \rightarrow \text{claroyderecha}, \{S\ is\ 0.1\} \quad (8)$$

Simplification could be applied to rule 8 using rule 7 and obtain rule 9 as follows:

$$s([and, light, R], S) \rightarrow \text{claroy}, p(R, SR), \{S\ is\ SR{\cdot}0.1\} \quad (9)$$

$$s([and, Q, R], S) \rightarrow p(Q, SQ), y, p(R, SR), \{S\ is\ SQ{\cdot}SR{\cdot}0.1\} \quad (10)$$

Now rule 9 could be simplified again, this time using rule 6, to obtain rule 10. Note that in this last rule, the connective is located at the corresponding position within the sentence. If now the agent invents or adopts a rule that associates the formula `[or,light,right]` with the sentence "clarooderecha" and applies simplification, then its grammar will contain the rule:

$$s([or, Q, R], S) \rightarrow p(Q, SQ), o, p(R, SR), \{S\ is\ SQ{\cdot}SR{\cdot}0.1\} \quad (11)$$

3.3.4. Adaptation

There is one more feature we will need to implement in our model which is commonly referred to as adaptation. This mechanism is a way to ensure that the agents' grammars are coordinated. It is likely to happen that different agents refer to the same boolean function in distinct ways, or that two agents choose to concatenate the expressions associated with the components of a given formula in different orders. Coordination is achieved through a process of self-organisation of the agents' linguistic interactions that takes place when the agents adapt their preferences for vocabulary and grammatical constructions to those they observe are used more often by other agents.

At the last step of a language game, when the speaker communicates its intended meaning to the hearer, the agents adapt the scores of their grammar rules. This happens only when the speaker can generate at least one sentence for the meaning it is trying to communicate and the hearer can parse the sentence produced by the speaker. In a language game only the agent playing the role of hearer adapts the scores of its grammar rules. However, all agents are fated (probabilistically speaking) to take both roles eventually, and thus have chances to adapt their grammars.

Whenever a game succeeds, the hearer takes the interaction as a positive instance and adjusts the scores of its grammar rules, both at interpretation and generation levels. The hearer also increases the scores of the rules it used to obtain the meaning that the speaker had in mind and decreases the scores of the grammar rules that generate competing meanings. After that, the hearer tries to express the meaning the speaker intended to communicate using its own grammar rules, and it increases the scores of the rules that generate the sentence chosen by the speaker and decreases the scores of the rules that generate competing sentences.

The process of adjusting the scores at the level of interpretation reduces ambiguity and discourages homonym sentences from appearing. The adjustment at generation level means reducing ambiguity and discouraging a phenomenon similar to synonymy of sentences. In any case, the scores of the grammar rules used by the hearer to obtain the meaning the speaker had in mind are increased only once.

If the meaning interpreted by the hearer is not logically equivalent to the meaning the speaker had in mind, the game fails, and the hearer decreases the scores of the grammar rules it used for obtaining the wrong meaning for the sentence uttered by the speaker.

3.4. This project in the literature

The project developed in this work belongs, as previously stated, to the evolutionary linguistics branch. More precisely, it fits more appropriately into the set of experiments that study grammar acquisition [BS13, GSB12, Vog05, vT12, SBK03, Kir02] rather than lexicon acquisition [Ste95, SB05].

It is worth noticing that word order plays a vital role in our experiments. This is because in the experiments described in [SSn14], which constitute the basis of the present work, the position of each sub-expression in a sentence determines how it is semantically related to the rest of sub-expressions. Kirby [Kir02] also studies the emergence of word-order based grammar, but without addressing the problem of negotiation, as the population he uses consists only of two individuals. In the experiments described in [SSn14], however, the population consists of 10 agents, which need to reach consensus on how to order the expressions associated with the constituents of each different type of boolean formula to construct a sentence. In the present project we wish to generalise this model so that it can use an arbitrary number of agents.

Another difference between the agent-based model described in [SSn14], on which the present work is based, and Kirby's work [Kir02] is the manner in which communicative success is evaluated. While Kirby considers syntactic equality as an indicator of mutual understanding, [SSn14] uses logical equivalence. As a result of this, the grammars constructed by the agents in the experiments described in [SSn14] do not impose a strict order between the expressions associated with the arguments of commutative boolean functions.

Beuls and Steels [BS13] also study grammar acquisition, but they use agreement markers instead of word order as the syntactic means for semantic disambiguation. Similarly to [SSn14], they perform simulations with software agents, initialise them with a predefined vocabulary for basic properties and use a language game in which the topic consists of several objects.

The differing point is that the meanings constructed by the agents in [BS13] consist of a set of distinctive properties in which each property refers to a single object of the topic. Moreover, the role of agreement markers is to indicate which properties of the distinctive set refer to the same object. Therefore, the set of meanings the agents can construct in the agent-based model proposed in [BS13] are only conjunctions of basic properties, whereas the agents in [SSn14] and those we intend to use in the present project conceptualise the topic by constructing a discriminating logical formula. These logical formulas can be a composition of basic properties through conjunction, disjunction, negation or any other boolean function which is true for every object in the topic and false for the rest of the objects in the context.

Finally, taking as a starting point the Prolog implementation of the agent-based model for studying the acquisition of a language system of logical constructions used in the experiments reported in [SSn14], we will build a set of Prolog software modules that allow implementing different types of agent-based models and conduct language evolution experiments with these models. These modules will be different from the Lisp simulation and grammar processing systems [Ste11a] used in most agent-based simulation experiments described in the literature [GSB12, Vog05, PH12, SB05, Ste95, Ste11a, vT12, SS12, BS13, Ste99].

4. The Prolog programming language

Before revealing the implementation details of the model developed in this project, it might be useful to give a brief overview of the Prolog programming language. To this effect, this chapter wants to discuss the basic terminology and concepts in Prolog and also the advanced techniques featured in the model. Nevertheless, this chapter is not a detailed explanation of all features in Prolog, and we strongly encourage the reader to refer to some suitable references in case necessary [CM03, SS94, Bra90]. Also, note that various of the technical aspects below explained have been inspired by [SS00].

4.1. Fundamentals and logic programming

To begin with, Prolog is a general purpose and logical programming language which has its roots in first-order logic. It is for this reason that writing Prolog programs can turn out to be a very different process compared to what one may encounter in other programming languages. Here, the user needs to specify which relationships and objects occur in the problem and which relationships are true about the desired solution. Therefore, the process is more about describing the problem as logical axioms rather than giving the sequential steps to solve it.

These kind of programs can be executed by providing the system with a problem, formalized as a logical statement and called the ***goal statement***. The execution is an attempt to prove the goal statement, given the assumptions in the logic program. For example, when we say "Sarah owns the laptop" we are declaring a relation of ownership between the objects "Sarah" and "laptop". In contrast, when we ask the question "Does Sarah own the laptop?" we are trying to find out whether or not that relation of ownership exists.

Thus, we are always working with a set of statements, and each statement is either a fact about given information or a rule about how the solution may relate to or be inferred from the given facts. In more technical terms, writing a Prolog program consists of specifying some of the following types of statements:

- ***Facts*** asserting that a relationship holds between objects.
- ***Queries*** asking whether or not a relationship holds between objects.
- ***Rules*** defining high-level relationships between objects.

Out of the former, facts are the simplest form of statements, and are all collected in a database maintained by Prolog. We now consider a small database containing the following facts, which relate 2 objects each:

```
likes(john,mary).
likes(john,book).
likes(mary,john).
```

The first fact states that an individual called John likes another individual called Mary. Names of individuals or objects are known as ***atoms*** and must always begin with a lowercase letter. Note that atoms can refer to any kind of objects and that the relationships are not reflexive (the first and third facts are not equivalent). As a remark for later, it is also common to use the term ***predicate*** to refer to relationships.

The second most simple form of statement in a logic program is a query. Queries are used to find whether a particular relationship exists and look exactly the same as facts (but can be easily told apart by the context). Answering a query means ***satisfying a goal*** and determining whether the query is a logical consequence of the program. In the examples shown next, Prolog will search the database for each formulated question and come up with an answer, which can either be positive or negative:

```
?- likes(john,money).
no
?- likes(mary,john).
yes
```

The `no` answer means that nothing unifies with the question. In our former case, it becomes clear that there is no such fact in the database that contains `john` and `money` as its arguments. In contrast, when a `yes` answer is returned, then there is a fact which has the arguments provided and in the same order. Prolog however, can do much more that answering yes or no questions, and it provides ways to make inferences from one fact to another. This implies that, for example, we can find out which objects does John like, instead of just recovering the same information we input beforehand.

To do this, we will need introduce the concept of ***logical variable***. A logical variable stands for objects that we are unwilling or unable to name. This conception is opposed to the idea of a ***constant***, which denotes a particular object such as an integer or an atom. Variables in Prolog are distinguishable from constants because they start with an uppercase letter.

A query containing a variable asks whether there is a value for the variable that makes the query a logical consequence of the program. This creates questions that look like "Does John like `X`?", `X` being a logical variable. We refer to a variable as ***instantiated*** when there is an object that the variable stands for, and as not instantiated when what the variable stands for is not yet known. The question considered in this paragraph will produce two answers, resulting from Prolog searching through all its facts to find an object that the variable could stand for and unifying the variable with it:

```
?- likes(john,X).
X = mary;
X = book;
no
```

After introducing variables, it is a good moment to discuss what is a ***term***, the single and broad data structure used in Prolog. To this effect, we provide the following recursive definition, appearing in [SS00]:

1. Constants and variables are terms.
2. An expression of the form $f(t_1, ..., t_n)$ is a term if f is a functor of arity n and each one of the t_i is a term. ***Functors*** are structures characterised by its name and the number of arguments they accept, known as ***arity***. The combination is conventionally indicated with the notation f/n.

Queries, goals and terms where no variables occur are called ***ground terms***. If a goal or query contains variables, those are existentially quantified. A query Q with the set of variables $\{X_1, ..., X_n\}$ is asking whether there are values for each one of the X_i such that the query holds (note that this may lead to several solutions, or even an infinite number of solutions). If Q had been a fact instead, then the meaning of the fact would hold for any possible combination of values in each X_i (and thus, the variables are implicitly universally quantified).

It is also possible in Prolog to form a ***conjunctive query***, which is a conjunction of goals $Q_1, ..., Q_n$ separated by commas. If any variable occurs in more than one of those goals, we refer to it as a ***shared variable***. Conjuctive queries are a logical consequence of the program if each one of the goals Q_i is also a consequence of the program and the shared variables are instantiated to the same values in all goals.

Finally, the third and most important statement is the rule, which allows to define new relations in terms of existing relations. Rules in Prolog contain a sequence of one or more goals (or as we have already seen, a conjunctive query). For instance, if we consider a rule of the form `H :- B,C.`, then `H` is syntactically the same as a literal and it is referred to as the ***rule head***, and the literals `B,C` form the ***rule body***. The symbol `:-` is the functor denoting a rule. This structure can be more informally read as `H` is true if `B` and `C` are also true. Note that facts are rules with a sequence of 0 goals, or in other words, no rule body.

To finish with the fundamentals of the Prolog programming language, we briefly explain the computational model of logical programs, which is based on a process called ***unification***. A unifier of 2 terms (or clauses) is a substitution of all variables X_i present with a term t_j such that it holds that every X_i does not occur in any of the t_j.

In the query `likes(john,X)` that we previously asked, when the literal `mary` is found, `X` becomes unifies with the atom `mary`. Prolog marks the place in the database where a unifier is found, in order to continue the search later on if necessary. Should the user or the program require more solutions, Prolog will resume the search from where it left the place marker to find another possible answer to the question. Returning to our example, `book` is the next literal in the database fulfilling the conditions. At some point, it will not be possible to re-satisfy the goal any more, and then the search stops and the question fails. When succeeding, the program provides a ***proof*** which gives the solution verifying the query.

The computation progresses via ***goal reduction***. At each stage there is some resolvent, that is a conjunction of goals to be proven. Prolog chooses a goal in the resolvent and a clause head that unifies with it from the logic program. A new resolvent is obtained by replacing the chosen goal with the body of the chosen clause and applying a unifier obtained from the head of the clause and the goal. Termination happens when the resolvent is empty.

Nevertheless, this backtracking behaviour of Prolog described can be altered, something that is necessary in various situations and also used to increase efficiency. This can be achieved by introducing the symbol `!`, known as ***cut***. Suppose that now we define the following rule in order to list all the objects that a certain individual likes, but we include a cut by mistake:

```
likings(X) :- likes(X,Y), write(Y), nl, !, fail.
?- likings(john).
mary
no
```

Had we not introduced a cut in the `likings` clause, then the program would have also written "book" after "mary", as it is another thing that John likes. The cut makes inaccessible markers for certain goals so that they cannot be re-satisfied. When the program encounters a cut, the effect is to commit the system to all the decisions made since the first clause was chosen. This includes the unification `Y = mary`. Because this decision cannot be altered, after writing "mary", and because `fail` does not allow the clause to succeed, the program tries to re-satisfy `likings`, which cannot be done because `X` is already instantiated in the question.

4.2. Definite Clause Grammars

A grammar for a language is a set of rules that specify what sequences of words are acceptable as sentences of that language. This means specifying how words must be put together into phrases and what orderings of phrases are allowed. Given a grammar for a language we can look at a sequence of words and see whether it meets the criteria for being an acceptable sentence. In this project, each agents builds its own grammar and the objective is that all grammars within the population are compatible.

A particularly simple kind of grammar is known as Context Free Grammar (CFG) [CM03]. To illustrate this concept, consider the following, which may be the start of a grammar for English sentences:

```
sentece --> noun-phrase, verb-phrase

noun-phrase --> determiner, noun

verb-phrase --> verb, noun-phrase

verb-phrase --> verb

determiner --> [the]

noun --> [apple]

noun --> [man]

verb --> [eats]

verb --> [sings]
```

The former grammar consists of a set of rules, and each one specifies a structure for the different kinds of phrases. For instance, the first rule indicates that a sentence needs to always be constructed using a noun-phrase followed by a verb-phrase. This process continues recursively, and now in order to know how to build a noun-phrase and a verb-phrase we need to refer to the subsequent rules. For example, the sentence "the man" is a valid noun phrase according to the second rule, "eats the apple" is a valid verb phrase according to the third rule, and the concatenation "the man eats the apple" is a valid sentence.

Formally, we refer to the left-hand side of the rules as the head and to the right-hand side as the body. Given a sentence and a certain rule, by repeatedly applying the rules whose head contains a symbol present in the body of the initial rule, we can parse sentences and determine whether the sentence is a valid statement of the type appearing in the first rule.

Definite Clause Grammar (DCG) [CKPR72] is formalism used in Prolog to represent grammatical knowledge. Prolog provides a particular grammar rule notation that is designed to help users building parsers, because it makes the code easier to read and suppresses irrelevant information. Although Prolog's grammar rule notation is self-contained, it is important to realise that it is only a shorthand for ordinary Prolog code, and that it is interpreted in this way. The actual notation is build around the notation of Context Free Grammars that we already introduced.

Grammar rules are Prolog structures with the main functor `-->`, which is declared as an infix operator and automatically translated by Prolog systems. For example, the grammar rule `sentence --> noun-phrase, verb-phrase` is automatically translated into a Prolog clause `sentence(S0,S) :- noun-phrase(S0,S1), verb-phrase(S1,S)`. This means that there is a sentence between `S0` and `S` if there is a noun phrase between `S0` and `S1` and a verb phrase between `S1` and `S`. In other words, the variables introduced keep consuming the input sequence, that is `S1` is the same as `S0` without a preceding noun phrase and seemingly, `S` is `S1` without a preceding verb phrase.

Terminal rules which introduce words are also automatically translated by Prolog. For instance, the grammar rule `determiner --> [the]` is transformed into a clause `determiner([the|S],S)`. After translating the entire grammar in this manner explained, we can ask Prolog to satisfy the goal `sentence(X,[])`, which will succeed if `X` is a valid sentence with respect to the grammar rules specified in the program. Note that the second argument is the empty list, which denotes that there is no remainder after parsing, or alternatively, the last position in `X`.

4.2.1. Extension of the grammar formalism

We have already discussed the basic principles of Prolog grammars, but it is worth noting that they do not need to be as restrictive as described up until now. It is possible to add extra arguments to phrase types. We have seen how an occurrence of a phrase type in a grammar rule translates to the use of a Prolog predicate with two extra arguments, but in fact, those predicates can have any number of arguments. Adding them can be useful in various situations, such as when trying to reconstruct complete parse trees out of the parse trees of each subcomponent. The rules in our model also use extra arguments for various reasons, which are explained later on in this section.

Also, up until now everything mentioned in the grammar rules had to do with how the input sequence is consumed. Every item in the rules has had something to do with those two extra argument positions that are added by the translator and every goal in the resulting Prolog clause has been involved with consuming some amount of the input. Sometimes we may want to specify goals that are not of this type, and the grammar rule formalism allows us to do this. Any goals enclosed inside curly brackets are to be left unchanged by the translator.

To illustrate this, we will use some grammar rules that appear in our model implementation. These rules take the form `p(M,S,U,Id,L) --> [w,o,r,d], {S is 1.0}`. Here `M` represents the meaning to communicate in prefix notation, `S` the score, `U` the number of uses this rule has undergone, `Id` the identifier of the rule, and `L` the list of identifiers of other rules used by this rule. Because this rule is a terminal one `L` should be instantiated to `[Id]`. Usually `M`, `U` and `Id` are also instantiated, and `L` grows by reconstruction during parsing.

Note that in the previous rule, `S` is inside curly brackets. This happens because the score has nothing to do with the input sequence and we would like to stop the translation mechanism from changing it. If we omit this fact, then Prolog translation will include new variables in the score computation and the resulting goal will probably be never satisfied.

By applying the induction operation of simplification, the agents can construct more complex grammar rules, such as `s([if,X,Y],R,0,Id,[Id|T]) --> [c,o,n,n], p(X,R1,_,_,L1), p(Y,R2,_,_,L2), {append(L1,L2,T), R is R1*R2*0.1}`. This grammar rule allows constructing conditional sentences by concatenating the word `[c,o,n,n]`, which expresses the boolean function `if`, to the expressions associated with two propositions `X` and `Y`, which correspond to the antecedent and the consequent of the meaning expressed by this grammar rule. As it can be observed, bracket notation is used to specify the part of the grammar rule that deals with the computation of the score and construction of the list of identifiers of the grammar rules used to generate a particular sentence.

4.3. Structure inspection

Because their definition is very comprehensive, terms can present a lot of different forms. Prolog has some defined predicates dedicated to recognise different types of terms, decompose them into their functor and arguments and create new terms. We use the term ***type predicates*** to denote those predicates that distinguish between different types of terms. Such predicates can test whether the given term is a structure or a constant, and even determine if the constant is an atom, an integer, etc.

An example that illustrates structure inspection is the predicate `functor(Term,F,Arity)`, which is true if `Term` is a term whose main functor has name `F` and arity `Arity`. This is important because it directly provides access to the functor name, arity and arguments of compound terms in a very simple way. Moreover, the system predicate `functor` can also be used to build a new term given a particular functor name and arity.

Another system predicate similar to `functor` is the one called `arg`, which accesses the arguments of a term rather than the functor name. The goal `arg(N,T,Arg)` is true if `Arg` is the `N`th argument of term `T`. This predicate can also be used in two ways: finding the argument of a given compound term and create a new term instantiating a variable argument of it.

There is also a system predicate called `univ`, denoted by the binary operator `=..`, that dissociates a term into a list containing the name of its functor followed by its arguments. The goal `Term =.. List` succeeds if `List` is a list whose head is the functor name of the term `Term` and whose tail is the list of arguments of `Term`. Like `functor` and `arg`, `univ` has two uses: it can either build a term given a list or a list given a term.

Structure inspection and creation predicates are used in the modules implementing the invention, induction and adaptation abilities of the agents, in particular when grammar rules are created or modified in some way by the agents. Because grammar rules are also Prolog terms, they can be accessed, decomposed and constructed using structure inspection predicates such as `functor`, `arg` and `univ`.

4.4. Meta-logical predicates

In this section we cover a certaint type of predicates, called meta-logical predicates [SS00], that are outside the scope of first-order logic because they query the state of the proof, treat variables as objects of the language (rather than the terms they denote), and allow the conversion of data structures to goals.

The most fundamental meta-logical predicate is `var(Term)`, which succeeds if the term `Term` denotes a variable. Note that here we are explicitly referring to a variable name, instead of its value. There is an opposite predicate to `var`, written as `nonvar(Term)`, which is only true if `Term` is not a variable.

When we place predicates such as `var` inside the body of a certain clause in order to decide whether the clause definition needs to be accessed or not by the program flow, we are performing a ***meta-logical test***. These tests refer directly to the current state of the computation and the values of certain terms. Possible applications of them include making the most appropriate choice of the goal order of clauses in a program or writing more straight-forward procedures.

Meta-logical type predicates can also be used to define more complicated meta-logical relations such as the predicate `ground(Term)`, which is true if `Term` is instantiated to a ground term, or elaborated unification algorithms that refine the system predicate `=/2` used in Prolog for unification of 2 terms.

By default, the predicate `=/2` does not enforce the ***occurs check***, which is a mechanism that causes the unification of a variable `X` with a term to fail if the term contains `X`. In order to implement this procedure in Prolog, one needs to be able to check whether two variables are identical (not unifiable, because any two variables can be unified). This test is a meta-logical one, and can be achieved by means of the predicate `==/2`. The goal `X==Y` succeeds if `X` and `Y` are identical variables (or identical constants) or two structures with main functors of the same name and arity, whose matching arguments `Xi` and `Yi` also verify `Xi==Yi`. Same as before, there is a system predicate with the opposite behaviour: `X\==Y`.

Finally, we would also like to mention the ability that Prolog has to make programs and data equivalent. There is a meta-logical system predicate that allows a term to be converted into a goal: the predicate `call(X)`, which calls the goal `X` for Prolog to solve. By the time it is called, the variable must be instantiated to a term or otherwise an error is obtained. This procedure allows meta-programming and is also crucial for defining negation and allowing the definition of higher-order predicates.

Meta-logical tests are used to implement the induction operator of simplification, which allows the agents to extract generalizations from the holistic grammar rules they invent or adopt as they play language games with other agents. These generalizations are incorporated into the agents' grammars in the form of lexical entries and grammatical constructions which contain variables, and therefore can be used to express larger subsets of meanings and to subsume other rules in their grammars.

4.5. Extra-logical predicates

In its pure logical programming paradigm, Prolog should be free of side-effects. However, there are a number of predicates able to induce certain side-effects when satisfied as logical goals, and we refer to them as the group of extra-logical predicates. Some examples are those predicates which deal with input/output operations, provide some interface with the underlying operating system or modify certain parts of the program itself. In this section, we focus on the latter of those aspects.

Each predicate defined by the user can either be a ***dynamic predicate*** or a ***static predicate***. The main difference is that the procedure of a dynamic predicate can be altered during the execution, while the procedure of static predicates remains fixed. Program access and manipulation predicates can be applied only to dynamic predicates.

To gain access to a program we can use the system predicate `clause(Head,Body)`. When calling this particular goal, `Head` must be instantiated. The behaviour obtained is that Prolog searches for the first clause in the program that unifies with `Head`. The head and body of this clause is then unified with `Head` and `Body`, and this succeeds once for each unifiable clause in the procedure.

After obtaining clauses in the program we can manipulate them in order to create different variations. However, we still need a way to add clauses to the program and remove them. The basic predicate for adding clauses is `assertz(Clause)`, which adds `Clause` as the last clause of the corresponding procedure. A variant of this predicate, `asserta` adds the clause at the beginning of a procedure.

Similarly, in order to delete clauses in the program we can use the predicate `retract(Clause)`, which removes the first clause unifying with `Clause`. The term `Clause` needs to be instantiated to the form `H :- B`, and the clause head `H` needs also to be instantiated.

It is worth mentioning that even though the predicates `assert` and `retract` introduce the possibility of programming with side-effects, it is in general considered bad programming practice to use them. A Prolog code abusing on side-effects is hard to read, debug and analyse. Nevertheless, under some circumstances, their use can be justified, for example, if a clause already follows logically from the program, it can be added as a way to gain efficiency. In turn, retracting redundant clauses increases speed by reducing the size of the program, and therefore the search time associated with each clause during unification.

In this project the predicates `clause`, `assertz` and `retract` are used, because the agents are building and refining their grammars during the execution as they play language games. The use of assertions is justified in the sense that the (dynamic) grammar rules added are the result of induction processes from rules already present in the agent's grammar. The retracting operation is also justified by the fact that the only grammar rules removed are the ones subsumed by more general rules added as a result of induction processes to the grammar.

4.6. Second-order programming

Prolog also includes some predicates that, instead of producing a single solution, provide sets of solutions as a solution. Finding all the instances of a query that are implied by a program is a not a first, but a second-order question, since it asks for the set of all the elements with a certain property. This is also not pure Prolog because in the logical paradigm all information about a certain branch of the computation is lost on backtracking. Predicates that return all the instances of a query are called ***all-solutions predicates***. Next, we describe two all-solutions predicates provided by Prolog [SS00].

The predicate `findall(Term,Goal,Sols)` is true if and only if `Sols` unifies with the list of values to which a variable `X`, not occurring in `Term` or `Goal`, would be bound by successive re-satisfaction of the conjunctive goal `call(Goal), X = Term`. Of course, this only makes sense if `Term` and `Goal` share some variables.

Procedurally, `findall(Term,Goal,Sols)` creates an empty list `L`, renames `Goal` to a goal `G`, and executes `G`. For each successful execution `G`, and until the predicate `G` fails, a copy of `Term` is appended to the list `L`. Finally, `Sols` is unified with `L`.

A similar predicate, `bagof(Term,G,Sols)`, behaves like `findall` except that if there are variables in `G` that do not occur in `Term` it returns through backtracking a list of solutions for each possible value of these variables. If there is no solution for `G` in the goal `bagof(Term,G,Sols)`, then the goal `bagof(Term,G,Sols)` simply fails.

The difference between `findall` and `bagof` is that in calls to `findall(Term,Goal,Sols)` all the instances of `Term` that represent solutions to `Goal` are collected together, regardless of possible different solutions for variables in `Goal` that are not shared with `Term`. Also, if there is no instance of `Term` that satisfies `Goal`, then the goal `findall(Term,Goal,Sols)` will succeed with `Sols = []`.

In short, first-order logic allows quantification over individuals. Second-order logic further allows quantification over predicates and uses rules with goals whose predicate names are variables. Thus, predicate names become "first-class" data objects that can be manipulated and modified. From an operational perspective, this implies that goals are constructed dynamically during the computation. We have already seen methods to deal with this in Prolog, such as the predicate `univ`, which can be used to construct a goal of the form `Goal =..[P|Xs], Goal`, where a variable representing a predicate `P` is applied to a list of arguments `Xs`.

Second-order all-solutions predicates are used to compute the sets of competing sentences and competing meanings that the agents require in order to adapt their preferences for vocabulary and grammar to those of the other agents, taking into account the results of the language games in which they participate during a simulation. Dynamic goal construction also plays an important role in the implementation of the induction operation of simplification, where parts of a grammar rule are reconstructed in terms of others.

5. Specification: modules and their interaction

During the process of writing a program, the necessity to provide a certain degree of documentation soon becomes self-evident. This is specially true when considering Prolog programs. One would like to provide enough hints so that external users can comfortably use the program without having to understand the implementation details. At the same time, other individuals wanting to dwell in the implementation and potentially improve it, may require some help to understand the underlying mechanisms.

One traditional manner of confronting this problem is providing comments in the very same code [Bra90]. Those comments should first explain what the program is, how to use it, and only then give details about the programming methods employed. In Prolog we are specially interested in discerning the top-level predicates and knowing their inputs and meaning, among other things.

With the name specification, we denote a document that explains the behaviour of a program sufficiently to achieve the former goals. This chapter provides a specification for the different modules of the model developed, following the method suggested in [SS94], which consists in covering the following items for each predicate:

- A ***procedure declaration*** stating the name and the arity.
- ***Type declarations*** of its arguments.
- A ***relation scheme***, that is a precise statement written in English or another language that explains the relation computed by the predicate.
- Details about ***usage modes***. Because of Prolog unification it is possible to define predicates that can be used in multiple manners. The specification needs to guarantee which uses are correct.
- ***Multiplicity of solutions***, which states the number of solutions of the predicate for each possible usage.

In addition to this information, the documented predicates will be grouped according to the modules in which they appear in, and we will also detail the interface of each module, in a similar way as the documentation of the Ciao system [BCC+06]. This information should enable the user to combine the modules when necessary and also use them independently.

Different modes are available in standard Prolog to describe the arguments of a predicate [SS94][BCC+06]. The convention is to use the symbol `+` for an instantiated argument (input value), the symbol `-` for an uninstanciated one (output value), `?` for either and `@` for an argument which is not further instantiated (not an output value).

It is worth noticing that different usages of the same predicate have arguments with different modes. For example, the standard predicate `append(Xs,Ys,Zs)` for concatenating the lists `Xs` and `Ys` to produce `Zs`, can either be used as `(+,+,-)`, `(-,+,+)` and even more variants.

5.1. Simulation

5.1.1. Module `language_game`

The `language_game` module is the one in charge of controlling the flow of the execution. For each language game to play, it randomly chooses two agents and a meaning to communicate. After the game is played, both speaker and hearer make modifications to their own grammars and statistics are collected. At the end of the simulation then, we are able keep track of the agents taking part in each game, the meanings communicated, the sentences used by the speaker, the hearer's interpretation of such sentences and whether there is agreement between the two.

5.1.1.1. Usage and interface

- **Library usage:**

  ```
  :- use_module(language_game).
  ```

- **Predicates:**
  - *Exported predicates:*

    ```
    main/1, roles_to_ports/4, report_ind/5
    ```

  - *Other relevant predicates:*

    ```
    last_resultS/1, last_resultC/1
    ```

- **Other modules used:**
  - *System library modules:*

    ```
    dec10_io [tell/1, told/0], format [format/2],
    random [srandom/1, random/3], read [read/2],
    sockets [connect_to_socket/3, socket_shutdown/2],
    system [current_host/1], write [write_canonical/2, write/1]
    ```

  - *Model modules:*

    ```
    agent [ag_socket_port/2, write_stream/2],
    conceptual_sys [meaning/1],
    population [players/3, ini_players/1, population_size/1],
    utils [rnd_select/3, atom_and_number/2]
    ```

5.1.1.2. Documentation on relevant predicates

`main/1` PREDICATE

`main(X)`

`X` is a list of atoms of length 3 and of the form `X = [NGames, NAgents, Step]`. The first element stands for the number of games in this particular simulation, and the second for the number of agents that will be used in it. The predicate runs the simulation using the specified number of rounds and agents, displaying the results and progression along the way. `Step` is the numerical distance between points in the graphical representations produced, and therefore determines the rate at which statistics are collected.

Usage 1: `main(+X)`

- *Description:* Should `X` be instantiated to a list of atoms of length 3, then the first element is assumed to be the number of games and the second element is assumed to be the number of agents. This goal performs a simulation according to the parameters given and writes the results upon ending. If `X` is instantiated to any other kind of term this predicate fails.
- *The following properties should hold at call time:* `X` must be instantiated to a compound term, this being a list of atoms of length 3.
- *The following properties should hold upon exit:* A simulation with the specified number of rounds and agents is executed.

`last_resultS/1, last_resultC/1` DATA PREDICATE

`last_resultS(NS), last_resultC(NC)`

`NS` and `NC` are integer numbers.

Usage 1: `last_resultS(-NS), last_resultC(-NC)`

- *Description:* `NS` and `NC` indicate the number of times any two agents participating in a certain window of past language games achieved communicative success and coherence respectively. These predicates are updated dynamically and counters are set to 0 every `Step` language games (please refer to `main/1` of this module). They are used to report statistical information at the end of the simulation.
- *The following properties should hold upon exit:* Both `NS` and `NC` are instantiated to integer numbers representing the number of language games played with communicative success and coherence between agents among the past `Step` games.

`roles_to_ports/4` PREDICATE

`roles_to_ports(Speaker, Hearer, SPort, HPort)`

`Speaker` and `Hearer` are integer numbers which act as identifiers for particular agents. This predicate, provides the port numbers `Sport` and `Hport` that allow to communicate with the speaker and hearer in a language game.

Usage 1: `roles_to_ports(+Speaker, +Hearer, -SPort, -HPort)`

- *Description:* `Sport` and `Hport` are the port numbers associated with the speaker and the hearer respectively. They can be used to exchange messages with the corresponding agent.
- *The following properties should hold at call time:* Both `Speaker` and `Hearer` are instantiated to integer numbers.
- *The following properties should hold upon exit:* `SPort` is instantiated to the port number used by `Speaker`, and `HPort` to the port number used by `Hearer`.

`report_ind/5` PREDICATE

`report_ind(Game, NGames, NA, Host, Step)`

This predicate is used to collect statistics during the execution. All arguments are integers, except `Host`. `Game` represents the current game, `NGames` the total number of games, `NA` the number of agents and `Step` the distance between plot points. `Host` is the qualified name of the current host in which the program is run.

Usage 1: `report_ind(+Game, +NGames, +NA, +Host, +Step)`

- *Description:* This predicate sends a message to all agents in the simulation via a socket connection, provided that `Step` games have passed since the last time the same message was send. Upon receiving this signal, the agents report the number of adoptions and inventions and write the values in appropriate files, accessible after the simulation has finished.
- *The following properties should hold at call time:* All arguments need to be instantiated to ground terms.

5.1.2. Module `population`

This module is responsible for choosing which agents should participate in an arbitrary language game. To do so, it maintains the number of times each pair of agents have played together, and uses that information to select the 2 most suitable individuals. Selection is carried out at random, but enforcing that all agents speak approximately the same number of times to each other. Therefore, after selecting a speaker, the hearer candidates who have not spoken with that speaker many times have a higher chance of being picked.

5.1.2.1. Usage and interface

- **Library usage:**
  `:- use_module(population).`
- **Predicates:**
  - *Exported predicates:*
    `ini_players/1, players/3, population_size/1`
- **Other modules used:**
  - *System library modules:*
    `random [random/3]`
  - *Model modules:*
    `utils [rnd_select/3]`

5.1.2.2. Documentation on relevant predicates

`ini_players/1` PREDICATE

`ini_players(N)`

`N` is an integer number.

Usage 1: `ini_players(+N)`

- *Description:* This predicate initialises an individual counter for each agent which monitors the amount of times that the agent has participated with each other agent in a language game. At the beginning, all the values are set to 0.
- *The following properties should at call time:* `N` is instantiated to the number of agents in the simulation.

`players/3` PREDICATE

`players(N, Speaker, Hearer)`

`N` is an integer type representing the `T-F` game of the simulation, where `T` is the total number of games and `F` the number of games already played. The predicate unifies the integers `Speaker` and `Hearer` with the identifiers of the two agents who will take part in the next language game, playing the roles of speaker and hearer respectively.

Usage 1: `players(+N, -Speaker, -Hearer)`

- *Description:* First of all, the integer `N` is used to select a speaker number `Speaker` in a rotational fashion, so that if the total population of agents is `P`, each agent acts as a speaker every `P` games. After choosing the speaker, this goal finds another number `Hearer` such that the represented hearer is the agent to which `Speaker` has spoken less times.
- *The following properties should hold at call time:* `N` must be instantiated.
- *The following properties should hold upon exit:* `Speaker` and `Hearer` are instantiated to integer numbers.
- *Multiplicity of solutions:* It can happen that there are several agents to which `Speaker` has spoken to less times. In such situations, one of them is chosen at random as `Hearer`.

`population_size/1` DATA PREDICATE

`population_size(P)`

`P` is an integer number.

Usage 1: `population_size(-P)`

- *Description:* This predicate can be used to obtain the population size, that is the number of agents `P` taking part in the simulation
- *The following properties should hold upon exit:* `P` is instantiated to an integer number.

5.2. Conceptual system

5.2.1. Module `conceptual_sys`

This module is the one in charge of defining and producing the logical formulas, or meanings, that the agents try to communicate when playing language games.

5.2.1.1. Usage and interface

- **Library usage:**
  `:- use_module(conceptual_sys).`
- **Predicates:**
  - *Exported predicates:*
    `alphabet/1, connectives/1, commutative_conns/1, meaning/1`
    `new_exp/1, propositions/1`
- **Other modules used:**
  - *System library modules:*
    `random [random/3]`
  - *Model modules:*
    `utils [rnd_select/3]`

5.2.1.2. Documentation on relevant predicates

`meaning/1` PREDICATE

`meaning(SM)`

The term `SM` is a list of 2 or 3 atoms representing a logical formula in prefix notation.

Usage 1: `meaning(-SM)`

- *Description:* This predicate generates a random meaning `SM` out of all the possible obtainable by combining the provided logical connectives, `connectives(C)` and propositions `propositions(A)`. The new meaning can be then used in the current language game.
- *The following properties should hold at call time:* The facts `propositions(A)` and `connectives(C)` should exist in the database.

- *The following properties should hold upon exit:* `SM` is instantiated to a list of 2 or 3 elements, depending on the number of arguments of the logical connective chosen. The first element is an atom representing a logical connective out of the ones provided, and the other elements are atoms representing propositions.
- *Multiplicity of solutions:* This predicate succeeds only once.

`new_exp/1` PREDICATE

`new_exp(E)`

`E` is a list of atoms which are all single characters and represents a random expression.

Usage 1: `new_exp(-E)`

- *Description:* This predicate generates an expression using the symbols in the alphabet provided, which is listed in `alphabet(Z)`.
- *The following properties should hold at call time:* The fact `alphabet(Z)` should exist in the database.
- *The following properties should hold upon exit:* `E` becomes instantiated to a list of 3 to 6 elements, each one an atom representing a particular character.
- *Multiplicity of solutions:* This predicate succeeds only once.

5.3. Generation and invention

5.3.1. Module `generation`

The `generation` module implements functionalities that allow agents acting as speakers in a language game to construct expressions which can be effectively communicated to the hearing agents. This is achieved using Prolog's parsing mechanism for Definite Clause Grammars, which on backtracking can be used to generate sentences from a given grammar. The present module also contains predicates that check the grammatical equivalence of two sentences expressing the same meaning with respect to a particular linguistic system and its associated conceptual system.

5.3.1.1. Usage and interface

- **Library usage:**

  ```
  :- use_module(generation).
  ```

- **Predicates:**
  - *Exported predicates:*

    ```
    coherence/4, first_max/2, generates/5
    ```

- **Other modules used:**
  - *System library modules:*

    ```
    dcg_expansion [dcg_translation/2], dynamic [dynamic/1],
    random [random/3], write [portray_clause/1]
    ```

  - *Model modules:*

    ```
    agent_params [games_played/1, initialise/0, new_rule_id/1],
    conceptual_sys [commutative_conns/1, new_exp/1],
    induction [simplify/1],
    utils [rnd_select/3, rnd_permu/2, flatten/2, subseq_rest/4]
    ```

5.3.1.2. Documentation on relevant predicates

`coherence/4` PREDICATE

`coherence(SMeaning, SExp, HExp, R)`

`SMeaning` is a list of atoms which represents a non-recursive propositional logic formula constructed from a fixed set of propositions and a single boolean function of one or two arguments, or a single atom representing an atomic propositional formula. This formula is the one the speaker tries to communicate in a language game. Both `SExp` and `HExp` are lists of characters representing the sentence used by the speaker to communicate the intended meaning `SMeaning` and the sentence the hearer prefers to communicate that same meaning. `R` is a boolean argument.

Usage 1: `coherence(+SMeaning, +SExp, +HExp, -R)`

- *Description:* This goal succeeds when `SExp` and `HExp` are grammatically equivalent expressions. We consider two arbitrary expressions to be grammatically equivalent if they use the same words to represent the boolean function and its propositions, and if the boolean function appears in the same position within the expressions and it is either commutative or the expressions associated to the propositions are also located in the same position in both expressions.
- *The following properties should hold at call time:* `SMeaning`, `SExp`, `HExp` should be instantiated to ground terms.
- *The following properties should hold upon exit:* `R` is instantiated to 1 when `SExp` and `HExp` are grammatically equivalent expressions and to 0 otherwise.
- *Multiplicity of solutions:* The solution obtained is deterministic and this goal succeeds only once.

`first_max/2` PREDICATE

`first_max(Set, Expressions)`

This predicate is used to obtain the sentence with highest score out of the possible options to express a meaning. `Set` and `Expressions` are both compound terms, more precisely sets containing expressions and scores.

Usage 1: `first_max(+Set, -Expressions)`

- *Description:* `Set` is a set of pairs `[Score, Expression]`, where `Score` is instantiated to an integer according to the score of the sentence `Expression`, which is a list of characters expressing a particular meaning. The element appearing on the head of the set `Expressions` is one of the sentences with higher score.
- *The following properties should hold at call time:* `Set` should be instantiated to a ground term.
- *The following properties should hold upon exit:* `Expressions` contains all the elements of `Set`, with an element of maximum score in the first position.
- *Multiplicity of solutions:* If there are several sentences with maximum score, one of them is chosen at random. This goal succeeds only once.

`generates/5` PREDICATE

`generates(Meaning, Expression, Rest, CanSay, Inv)`

`Meaning` is a list of atoms which represents a non-recursive propositional logic formula constructed from a fixed set of propositions and a single boolean function of one or two arguments. `Expression` is a list of characters that corresponds to a sentence the agent in question can use to communicate `Meaning` to other agents. `Rest` is a set consisting of pairs of the form `[Score,Sentence]`, where `Sentence` is any sentence the agent could generate to communicate `Meaning` by using its grammar and `Score` a numerical value representing the score of the corresponding sentence. `CanSay` is a boolean value that indicates whether the agent can construct a sentence to communicate `Meaning` using the vocabulary and grammatical constructions stored in its grammar at call time. `Inv` is a boolean value which indicates if the agent had to invent a new sentence to communicate `Meaning`, because it could not express it using the vocabulary and grammatical constructions it knew at call time.

Usage 1: `generates(+Meaning, -Expression, -Rest, -CanSay, -Inv)`

- *Description:* `CanSay` indicates if the agent can construct a sentence to communicate `Meaning` using the vocabulary and grammatical constructions in its grammar at call time. If `CanSay` is 1, `Expression` is instantiated to one of the sentences with highest score out of the ones the agent can construct to express `Meaning`, `RestExp` is the set of all pairs `[Score,Sentence]` the agent can construct to communicate `Meaning` using its grammar at call time and `Inv` is 0. If `CanSay` is 0, `Expression` is instantiated to a sentence invented by the agent to communicate `Meaning`, `RestExp` is not instantiated and `Inv` is 1.
- *The following properties should hold at call time:* `Meaning` should be instantiated to a ground term.
- *The following properties should hold upon exit:* If `CanSay` is 1, `Expression` is instantiated to a list of characters which corresponds to one of the sentences with highest score the agent can construct to communicate `Meaning` and `Rest` to the set of all pairs `[Score,Sentence]` it can construct to express `Meaning`. Otherwise, `Expression` is instantiated to a sentence invented by the agent to communicate `Meaning` and `Rest` is not instantiated. `Cansay` and `Inv` are always instantiated to boolean values.

- *Multiplicity of solutions:* If `CanSay` is 1 and there are several sentences with highest score that the agent can construct to communicate `Meaning`, `Expression` is instantiated randomly to one of them. If `CanSay` is 0, `Expression` is instantiated to a sentence invented by the agent, which is also generated by a random process.

Usage 2: `generates(+Meaning, +Expression, -Rest, +CanSay, +Inv)`

- *Description:* This goal succeeds if `Expression` is the single sentence with the highest score the agent can construct to express `Meaning` using the vocabulary and grammatical constructions in its grammar at call time.
- *The following properties should hold at call time:* `Meaning` and `Expression` should be instantiated to ground terms, `Cansay` should be 1 and `Inv` should be 0.
- *The following properties should hold upon exit:* This goal succeeds if `Expression` is the single sentence with highest score that the agent can construct to express `Meaning` using its grammar. It fails if the agent cannot construct any sentence to express `Meaning`, or if it can construct some sentence with higher score than `Expression`. It may also fail if there are several sentences with highest score and `Expression` is just one of them.

5.4. Interpretation and adoption

5.4.1. Module `interpretation`

The present module implements functionalities that allow agents acting as hearers in language games to parse the expressions uttered by the speakers. In broad terms, the hearer tries to infer a meaning based on the expression received, and then this is compared to the correct meaning the speaker was intending to communicate.

5.4.1.1. Usage and interface

- **Library usage:**
  `:- use_module(interpretation).`
- **Predicates:**
  - *Exported predicates:*
    `understands/7`
- **Other modules used:**
  - *System library modules:*
    `dcg_expansion, dynamic [dynamic/1], lists [delete/3]`
  - *Model modules:*
    `agent_params [new_rule_id/1],`
    `conceptual_sys [commutative_conns/1],`
    `generation [generates/5, first_max/2],`
    `induction [simplify/1], utils [clean_body/2]`

5.4.1.2. Documentation on relevant predicates

`understands/5` PREDICATE

`understands(Exp, SMeaning, HMeaning, Rest, CanUnd, Agree, Adopt)`

`Exp` is a list of characters representing the expression used by the speaker to communicate the meaning `SMeaning`. `SMeaning` is a list of atoms which represents a non-recursive propositional logic formula constructed from a fixed set of propositions and a single boolean function of one or two arguments. `HMeaning` is the meaning obtained by the hearer by parsing expression `Exp` using the vocabulary and grammatical constructions in its grammar. This predicate is implemented using Prolog's parsing mechanism for Definite Clause Grammars. `Rest` is a list of other possible meanings which are also plausible according to the rules rules in the hearer's grammar. `CanUnd` is a boolean value that determines whether or not the hearer could parse `Exp` or not. `Agree` is a boolean value that indicates if the meaning `HMeaning` inferred is correct, that is, if it is logically equivalent to `SMeaning`. Finally, the argument `Adopt` is also of boolean type and becomes 1 when it is not possible for the hearer to infer the meaning of `Exp`, because there are no rules that allow the parsing of the expression given.

Usage 1: `understands(+Exp,+SMeaning,-HMeaning,-Rest,-CanUnd,-Agree,-Adopt)`

- *Description:* There are two possibilities for this goal to succeed, which are mutually exclusive and jointly exhaustive. In the first situation the hearer is able to use its grammar rules to interpret `Exp`, which is the expression uttered by the speaker, and obtain a meaning `HMeaning`. The formula `HMeaning` is the one with highest score out of all possible interpretations of `Exp`. The arguments `CanUnd` and `Adopt` take the values 1 and 0 respectively, because the hearer could interpret the given expression without the need of adopting any new rules. `Agree` is 1 if `SMeaning` and `HMeaning` are logically equivalent, and 0 otherwise.

  The other possible scenario is that the hearer is unable to interpret the expression `Exp`. In this case `HMeaning` is instantiated to the same as `SMeaning` and both `CanUnd` and `Agree` are instantiated to 0. When this happens, the hearer adopts a new holistic rule which matches `Exp` to `SMeaning`, and this rule is added to its own grammar and simplified. Because of this, `Adopt` is instantiated to 1, as the hearer established a new connection between the given expression and the meaning the speaker intended to communicate.

- *The following properties should hold at call time:* `Exp` and `SMeaning` need to be instantiated to ground terms.

- *The following properties should hold upon exit:* `HMeaning` is instantiated to the meaning the hearer inferred from `Exp`, or `SMeaning` if a new rule is adopted. `Rest` is instantiated to the list containing all other possible meanings for the given expression, only if `CanUnd` is true. `CanUnd` becomes 1 or 0 depending whether the speaker could interpret `Exp`. `Agree` becomes 1 if the inferred meaning is logically equivalent to `SMeaning` and 0 otherwise. The argument `Adopt` is instantiated to 1 if the hearer adopted a new rule to its grammar, and 0 if it did not.

- *Multiplicity of solutions:* It is possible that several valid meanings with high score exist. In this case the value for `HMeaning` is randomly chosen among them, in a non-deterministic manner.

5.5. Adaptation

5.5.1. Module `adaptation`

The agent acting as hearer learns from the outcomes of the languages games and modifies the rules in its grammar in order to be more successful in the future. This process is achieved by altering the weights of certain grammar rules involved in the parsing of the expression or competing rules with the ones used, as described in chapter 3. The overall process and functionality is implemented in this module.

5.5.1.1. Usage and interface

- **Library usage:**
  `:- use_module(adaptation).`
- **Predicates:**
  - *Exported predicates:*
    `repair_scores/5`
- **Other modules used:**
  - *System library modules:*
    `dynamic [dynamic/1]`
  - *Model modules:*
    `conceptual_sys [commutative_conns/1, propositions/1], generation [generates/5], utils [clean_body/2]`

5.5.1.2. Documentation on relevant predicates

`repair_scores/5` PREDICATE

`repair_scores(Exp, HMeaning, Rest, CanUnd, Agree)`

`Exp` is a list of characters representing the expression used by the speaker. `HMeaning` is a list of atoms which represents a non-recursive propositional logic formula constructed from a fixed set of propositions and one boolean function of one or two arguments or a proposition by itself.

This expression is the one the hearer interpreted from `Exp` using its grammar rules. `Rest` is a list of other possible interpreted meanings. `CanUnd` is a boolean value that indicates whether or not the hearer could infer a meaning for `Exp` by itself or not. `Agree` is a boolean value that indicates if the meaning `HMeaning` inferred by the hearer is correct, that is, it is logically equivalent to the meaning the speaker had in mind.

Usage 1: `repair_scores(+E,+HMeaning,+RMeaning,+CanUnd,+Agree),`

- *Description:* This goal succeeds with no effect in case `HMeaning` is a propositional formula because the agents already have a common vocabulary for propositional symbols or basic categories.

  When `HMeaning` is a more complex formula, and in case `CanUnd` and `Agree` are both true, then this predicate reinforces the grammar rules that the hearer used to obtain `HMeaning`. It also discourages grammar rules used to obtain competing meanings. Next, it also discourages rules in the hearer's grammar that generate competing sentences different from `E` for expressing `HMeaning`.

  Finally, when `Agree` is false, then this predicate only discourages the rules used by the hearer to obtain the meaning `HMeaning`. In case there is no agree nor coherence nothing happens.

  Adaptation is done by altering the scores inherent to the affected rules, encouraging following the update $S = S \cdot (1 - \mu) + \mu$ and discouraging $S = S \cdot (1 - \mu)$, where S is the score of the rule in question. The parameter μ controls the speed at which adaptation occurs, and it is called the alignment rate. In our case, it takes the value 0.1 by default.

- *The following properties should hold at call time:* All variables need to be instantiated to ground terms.

- *The following properties should hold upon exit:* Rules are updated and feedback written in the user interface, according to the behaviour given in the description of this goal.

- *Multiplicity of solutions:* This goal succeeds with a deterministic result.

5.6. Induction

5.6.1. Module `induction`

This module is responsible for implementing the induction capabilities of the agents. With those, the different participants in language games are able to effectively learn and generalise their rules. Induction is applied when an agent invents a new rule or adopts one from another agent. In our case, we only use the simplification operator as described in chapter 3.

5.6.1.1. Usage and interface

- **Library usage:**
  `:- use_module(induction).`
- **Predicates:**
  - *Exported predicates:*
    `simplify/1`
- **Other modules used:**
  - *System library modules:*
    `dcg, dcg_expansion, dynamic, terms, write`
  - *Model modules:*
    `agent_params [new_rule_id/1], utils [clean_body/2]`

5.6.1.2. Documentation on relevant predicates

`simplify/1` PREDICATE

`simplify(R)`

`R` is a well-formed Prolog grammar rule which has been recently invented by an agent or adopted by an agent from another agent.

Usage 1: `simplify(+R)`

- *Description:* This predicate tries to generalise the given rule by creating a new one which introduces variables to represent propositional symbols. This is done using the simplification mechanism already discussed. Given two different rules R and $r2$ with $r2$ being propositional and provided that the expression in $r2$ is a substring of the expression in R, then simplification can be applied to R by replacing the symbol in the meaning with a free variable, and replacing the substring in R with a non-terminal.
- *The following properties should hold at call time:* `R` is instantiated to a grammar rule.
- *The following properties should hold upon exit:* If simplification could be applied to the rule `R` provided, then `R` is replaced by the simplified rule in the agent's grammar.

5.7. Agents

5.7.1. Module `agent`

The `agent` module is the one in charge of simulating the behaviour of each agent participating in the simulation. It implements predicates which allow the agents to respond to incoming messages when acting as hearers and to speak to other agents when acting as speakers. This module is also responsible for collecting various information, such as the number of inventions and adoptions per agent or the agent's inner grammar. This information is used at the end of the simulation to produce some statistics indicators of the overall process.

5.7.1.1. Usage and interface

- **Library usage:**

  `:- use_module(agent).`

- **Predicates:**
  - *Exported predicates:*

    `main/1, ag_socket_port/2, write_stream/2`

- **Other modules used:**
  - *System library modules:*

    `concurrency [concurrent/1, eng_call/3], dynamic [dynamic/1],`
    `format [format/2], random [srandom/1],`
    `read [read/2], sockets [bind_socket/3, socket_accept/2],`
    `system [current_host/1], write [write/2, write_canonical/2]`
  - *Model modules:*

    `adaptation [repair_scores/5],`
    `agent_params [games_played/1, update_games_played/0, ...`
    `... initialise/0], generation [generates/5, coherence/4],`
    `interpretation [understands/7],`
    `utils [atom_and_number/2, clean_atom/2, clean_body/2]`

5.7.1.2. Documentation on relevant predicates

`main/1` PREDICATE

`main(X)`

`X` is a list of 2 arguments of the form `X = [AgentNumber, Seed]`. Here, `AgentNumber` is a single integer and it is understood as the identifier of the agent. `Seed` is an integer used as a seed for the random Prolog generator (in our case, it is computed automatically). Generally speaking, the behaviour of this predicate is to load the agent's initial rules for expressing propositions and prepare the agent to read and respond to messages.

Usage 1: `main([+AgentNumber])`

- *Description:* This predicate tries to satisfy in a separated engine stack (potentially a new thread) a routine that reacts to messages sent to this agent in either the roles of speaker or hearer. As a speaker, the expression generated to communicate a certain logical formula is sent back and to the agent acting as a hearer.

As a hearer, the agent tries to interpret the expression received and updates its rules accordingly, depending on whether the inferred formula agrees with the one the speaker intended to communicate or not. This predicate also initialises the common vocabulary for each agent and data predicates needed for invention and adoption statistics.

- *The following properties should hold at call time:* `Agent number` must be instantiated to a ground term.

`ag_socket_port/2` PREDICATE

`ag_socket_port(Port, NA)`

Both arguments are integers. `Port` indicates the number of the port associated to the agent with identifier `NA`.

Usage 1: `ag_socket_port(+Port, -NA)`

- *Description:* Obtain the identifier `NA` of the agent linked to port `Port`.
- *The following properties should hold at call time:* `Port` is instantiated to a ground term.
- *The following properties should hold upon exit:* `NA` is instantiated to an integer representing the agent associated to the port.

Usage 2: `ag_socket_port(-Port, +NA)`

- *Description:* Obtain the port number `Port` linked to the agent with identifier `NA`.
- *The following properties should hold at call time:* `NA` is instantiated to a ground term.
- *The following properties should hold upon exit:* `Port` is instantiated to an integer representing the port associated to the agent.

`write_stream/2` PREDICATE

`write_stream(Stream, Term)`

`Stream` is a Prolog stream and `Term` is any term to write in it.

Usage 1: `write_stream(+Stream, +Term)`

- *Description:* The term `Term` is written to the `Stream`.
- *The following properties should hold at call time:* Both arguments are instantiated at call time.

5.7.2. Module `agent_params`

This module defines several parameters that are part of the implementation of each agent in the population. The actual values of these parameters however, should be maintained individually by each agent, because they depend on the interaction history of each agent, which is different from the interaction histories of the rest of the agents in the population.

First, it monitors the current number of the game being played by a particular agent. It also maintains the identifiers of past grammar rules created by this agent, so that the agent can select an appropriate identifier when adding new rules or inducing. Finally, it also implements a predicate that needs to be called prior to the start of the simulation and loads the rules for the propositional vocabulary and the initial variables needed.

5.7.2.1. Usage and interface

- **Library usage:**

  ```
  :- use_module(agent_params).
  ```

- **Predicates:**
  - *Exported predicates:*

    ```
    games_played/1, initialise/0, new_rule_id/1,
    update_games_played/0
    ```

- **Other modules used:**
  - *System library modules:*

    ```
    dcg_expansion [dcg_translation/2], dynamic[dynamic/1]
    ```

  - *Model modules:*

    ```
    conceptual_sys [new_exp/1, propositions/1]
    ```

5.7.2.2. Documentation on relevant predicates

`games_played/1` DATA PREDICATE

`games_played(N)`

`N` is an integer value.

Usage 1: `games_played(-N)`

- *Description:* This is a changing data predicate which provides the number of games played by a particular agent at a certain point.
- *The following properties should hold at call time:* The fact `games_played(N)` should exist in the database.
- *The following properties should hold upon exit:* `N` is the number of the last game played.
- *Multiplicity of solutions:* This predicate succeeds only once.

`initialise/0` PREDICATE

`initialise.`

This predicate describes an initialisation process conducted by all agents prior to the start of the simulation itself.

Usage 1: `initialise.`

- *Description:* Upon calling, the rules that allow the agents to express the common vocabulary for basic categories (which is assumed to be agreed on before starting the simulation) are created. The predicate also initialises the data predicates counting the number of rules created, its date and the number of games.
- *The following properties should hold upon exit:* Data predicates representing number of games, rule identifiers and rule dates are added to the database, together with the initial grammar rules on basic propositions. The agent is in an appropriate state to start the simulation.

`new_rule_id/1` PREDICATE

`new_rule_id(Id)`

`Id` is an integer number representing the identifier that must be hold by the next rule asserted during the simulation.

Usage 1: `new_rule_id(-Id)`

- *Description:* `Id` is instantiated to the integer of the next rule identifier to be used, the one after the last rule created. This goal also sets the date of this new rule to the number of the current game.

- *The following properties should hold at call time:* The data predicates `rule_id/1` and `games_played/1` must exist in the database. Their arguments should be integers representing the first free identifier for a rule and the number of games played respectively.
- *The following properties should hold upon exit:* The fact `rule_id(Id)` is retracted and a new fact `rule_id(Id1)` is added, `Id1` being `Id+1`. A fact `rule_date(Id,D)` is also added, where `D` is the number of the current game.

5.8. General functionality

5.8.1. Module `utils`

This module implements various predicates that are of general interest. Typically, these are used by various modules and provide certain operations on data structures. Those predicates are not relevant to the user and not necessary to understand how the present model works, but we decided to list them here regardless for reference.

5.8.1.1. Usage and interface

- **Library usage:**

  ```
  :- use_module(utils).
  ```

- **Predicates:**
  - *Exported predicates:*

    ```
    atom_and_number/2, clean_body/2, flatten/2, insert_at/4
    remove_at/4, rnd_permu/2, rnd_select/3, subseq_rest/4
    ```

- **Other modules used:**
  - *System library modules:*

    ```
    random [random/3]
    ```

5.8.1.2. Documentation on relevant predicates

`clean_body/2` PREDICATE

`clean_body(Body, BodyC)`

`Body` is instantiated to the body of a Prolog grammar rule. `BodyC` is of the same form but without the module prefixes added by the Ciao system.

Usage 1: `clean_body(+Body, -Body2)`

- *Description:*: This predicate strips the prefixes 'dao_chunk:', 'multifile:', 'lists:' and 'iso_misc:' from `Body`. To clean a rule body we first separate each literal in it. A literal representing the score constructed with the infix operator `'is'` needs no cleaning. For all other literals, we isolate the functor and keep stripping the prefixes added to it until there are none remaining.
- *The following properties should hold at call time:* `Body` must be instantiated to the right-hand side of a Prolog grammar rule.
- *The following properties should hold upon exit:* `BodyC` is instantiated with the same content and structure as `Body`, but stripping any prefixes of the form 'dao_chunk:', 'multifile:', 'lists:' or 'iso_misc:'.

6. Case studies and experimental results

The model described and implemented in this project has been validated by conducting a series of experiments that study both the emergence of a shared language system of logical constructions and its transmission from one generation to the next. Our model's behaviour in both of these scenarios is discussed in the subsequent sections.

The results that we are going to analyse are produced automatically by the program at the end of the simulation. After a series of runs with the parameters specified by the user, the model developed outputs a graphical representation that shows how different indicators evolve within the population and the final grammars for each agent. The indicators we use to describe the behaviour of the population are as follows:

- Communicative success: Proportion of language games completed successfully.
- Coherence: Proportion of language games in which the hearer correctly understood the sentence communicated and in which the hearer would produce the same sentence if it had to communicate the meaning the speaker had in mind itself.
- Adoption: Number of sentences adopted.
- Invention: Number of sentences invented.

It is worth noting that the first two indicators are computed using a certain discretisation step. Therefore, communicative success and coherence at an arbitrary point in a plot are the average over a certain number of past language games. In contrast, adoption and invention are calculated as the average values over all agents in the simulation and cumulatively since the first language game played.

6.1. Emergence of a language system

Experiments to study the emergence of a shared language system follow exactly the mechanisms described in chapter 3. At the beginning, agents in the simulation possess a common lexicon of six basic categories but no grammar rules. The agents start then to play language games using the logical formulas derived from combining the former categories with one unary or binary boolean connective.

A possible outcome is shown in figure 6.1, which uses a population of 10 agents and simulates 3600 language games. It can be seen that communicative success increases swiftly, reaching its peak (1.0) after 2385 games and not decreasing from that point onwards. Coherence also reaches its maximum value and stabilises after 2385 games, but generally shows a slower progression

than communicative success, for the values of the latter remain practically 1.0 since game 975. In turn, the number of adoptions show a steep increase during the first hundreds of language games, before reaching its maximum value of 29.32 at game 1560 and remaining constant until the end of the simulation. Inventions evolve in a curve with a similar fashion, attaining a maximum of 5.62 at game 315.

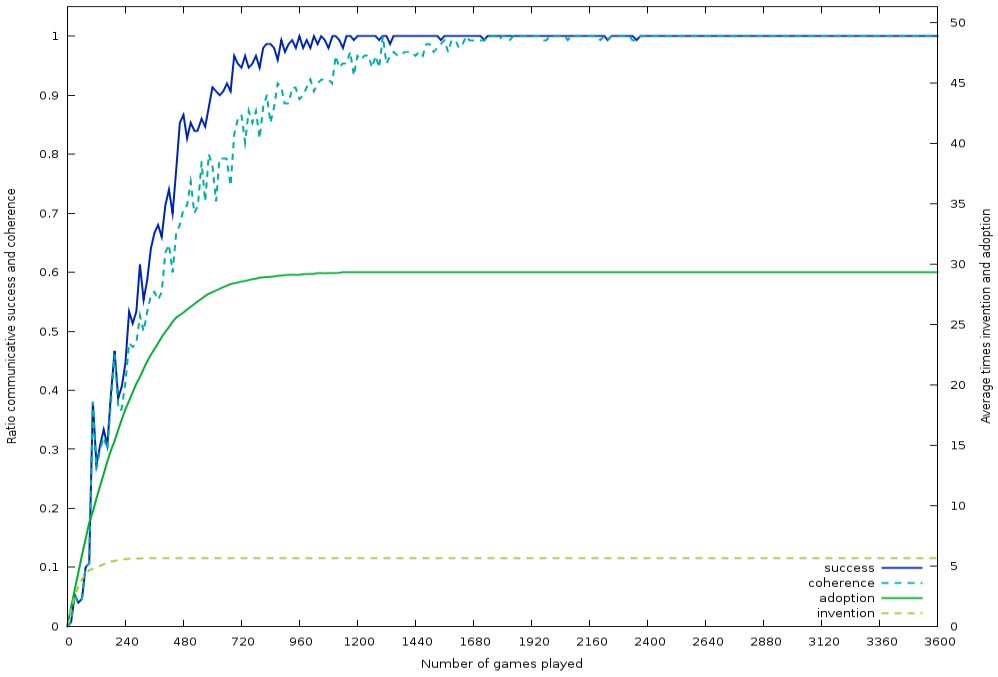


Figure 6.1: Language system emergence experiment with a population of 10 agents playing 3600 language games. The results are averaged over 10 executions and using a step of 15 games.

The former results provide some insight into the behaviour of the model developed. Increasing communicative success and coherence show that a common shared language allowing agents to communicate has risen. It is also expected for the number of sentences adopted and invented to reach a point of stagnancy. In the first case, this happens when all the agents have learnt the vocabulary and grammatical constructions used by the other agents. In the second case, it means that the agents have learnt all the vocabulary and grammatical constructions required to express any logical formula. It seems reasonable that the latter occurs earlier on during the simulation, that is to say agents can easily obtain an universal set of vocabulary and rules, but the truly difficult task is for this set to be shared by (or equivalent to) the ones used by other agents.

Nevertheless, it can be argued that an experiment with only 10 agents is far from realistic. In another simulation shown in figure 6.2, we performed a similar experiment but this time using a 5 times larger population of 50 agents, who play 18000 language games. While the number of agents is still small, it may account for a small community in real life. In this case, full communicative success is reached at game 15600 and full coherence is never achieved, although it is invariably over 0.98 and increasing since game 13000. Even more language games would be needed in this setting for perfect coherence. Adoptions do not stabilise either, and they keep growing very slowly over 80.8. Inventions cease at game 2400 with an average value of 5.6.

What it is important to observe here by comparing the first and second experiments is that time required for full coherence and communicative success grows significantly with respect to an increase in the population. This is because in order to achieve a common language, agents need to self-organise by speaking to each other many times, and the complexity of such a feat increases combinatorially with the number of agents. For the very same reason the amount of adoptions increases with respect to the former experiment. The number of inventions and its evolution however, remains approximately the same, which is explained because the number of logical connectives and propositions has not changed. Once the agents have invented or adopted grammar rules to express all possible meanings, there is no need to invent any more sentences. Also, there is no need either to invent already adopted rules for a particular meaning.

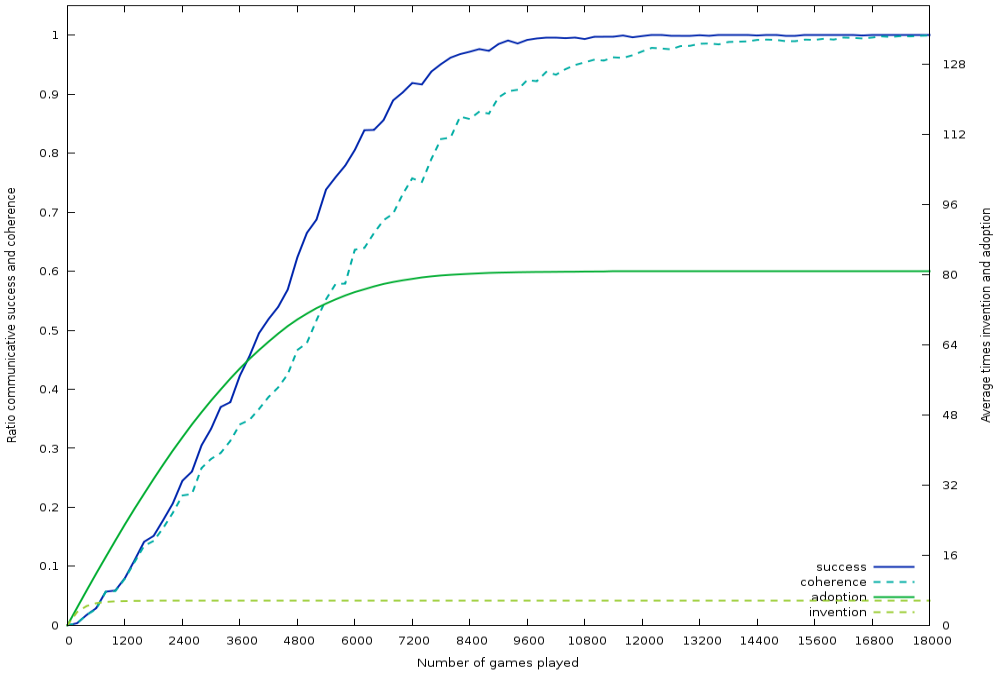


Figure 6.2: Language system emergence experiment with a population of 50 agents playing 18000 language games. The results are averaged over 10 executions and using a step of 200 games.

Apart from the previous plots, the program also writes the final grammars for each participating agent, which can be consulted by the user when the simulation is finished. These indeed show that we are obtaining sensible results and that the induction and adaptations mechanisms are working as expected.

A grammar rule constructed by one of the agents in a particular simulation run of the emergence experiment is shown next as an example of the former. It states that in order to construct a meaning of the form `[or,D,E]`, where `D` and `E` are certain propositions, this particular agent can append a prefix "rev" (denoting the `or` connective), the word for `E` and the word for `D` in this order. Moreover, this rule is highly preferred by this agent because of its score weight of approximately 0.94. The score `A` of a sentence generated using this rule is computed as the product of 0.94 times the scores of the propositional rules, `J` and `M`, which are computed recursively in the same manner.

```
s([or,D,E],A,28,47,[47|F],B,C) :-
    'C'(B,r,G),
    'C'(G,e,H),
    'C'(H,v,I),
    p(E,J,_,_,K,I,L),
    p(D,M,_,_,N,L,C),
    append(K,N,F),
    A is J*M*0.9449220687198503.
```

6.2. Transmission of a language system

In the previous section, we have seen that our model is indeed capable of producing a shared language via the self-organisation of the agents. This result assumes however, that the agents can play language games indefinitely. In real life, the population is ever-changing and people are born and die. Moreover, most individuals do not attain a perfect command of the language.

A simplified form of this circumstance can be simulated with the model developed as well. To do this, we introduce in this section a variant to the previous experiment. The agents are now divided into 3 equality sized groups: the elder, the adults and the young. They keep playing language games under similar conditions as before, but now, after a certain number of language games, a generation shift occurs. This is to say, the elders die, the adults become elders and the young become adults. Additionally, a new group of agents are born to replace the fallen elders, but their grammars are completely empty (except for basic categories) and thus they have no knowledge of the language. With this setting we aim to see how a language system is passed from one generation to the next.

Because this is a another type of language game, we needed to introduce a new module to replace `language_game`, called `language_game_trans`. Nevertheless, owing to the independent modular structure of the model, we were able to reuse all other modules. Overall, the main difference between `language_game` and its variant is that the agents are divided into generational groups, which are alternated after a number of simulations. The agents acting as elders are replaced by new agents who only know the vocabulary for basic categories. After 3 generation shifts the population is completely renewed, while the learning process remains constant.

Figure 6.3 shows the results obtained under these circumstances, with a population of 10 agents playing 4200 language games and generation shifts occurring every 500 language games. Each time a new generation is introduced the four measures drop drastically, but they catch up before the next generation of agents takes over. It is clear that the agents do not gain full communicative success, although its values remain reasonably high at the peaks and typically reach 0.97 right before a generation shift. Coherence follows closely with values of 0.95 or higher, displaying a slower progression as previously discussed but now more attenuated as generations ensue.

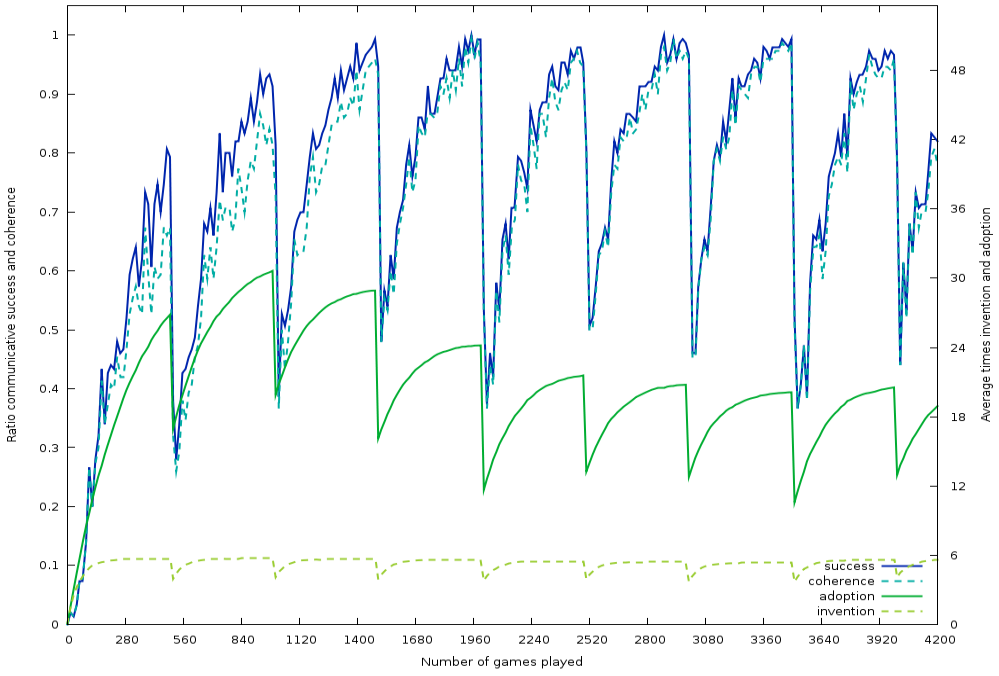


Figure 6.3: Language system transmission experiment with a population of 10 agents playing 4200 language games. Generation shifts occur every 500 language games. The results are averaged over 10 executions and using a step of 15 games.

Invention shows small fluctuations when a generation shift occurs, but overall its values are very similar to the ones in the emergence experiments, between 5.4 and 5.7. Adoption follows a similar fashion and has a smaller average value of approximately 20. This is due to the fact that new agents learn an already established language in the population, the language that is transmitted from one generation to the next, whereas agents created in the emergence experiments must learn all the constructions invented by the rest of the population. This partially established language uses fewer variations (typically 1 or 2) for expressing a given meaning.

In this new experiment, words are only invented by the young agents, who do not have any knowledge of the language but however have the need to express some meanings when playing language games. These invented words are often neglected by the other agents who will probably teach to the young the broadly accepted sentence for expressing this particular meaning. Nevertheless, in the long run it is possible that some of these words take over and become accepted by the rest of the population. This happens in real life as well, where we often see younger generations coining neologisms which can at times pose difficulties to the rest of the population.

We now discuss the effects of using a larger population with the same type of language game. In this case we used 50 agents playing 19200 language games and with generation shifts occurring every 3800 games. The results obtained are shown in figure 6.4. It can be seen that before the first shift, the agents did not have time to learn all the language and communicate effectively. Thus, success and coherence values remain low and well under 0.5. Regardless, after the new generation is introduced and the unavoidable drop in success and coherence occurs, the population is able to overcome this situation and obtain a higher values of 0.9 and 0.75 for communicative success and coherence respectively before the following generation shift.

As happened before in the smaller version of the experiment, communicative success and coherence keep increasing and reach higher values after each generation and before the next one, although now this effect is more visible. This is connected with a drop in the number of adoptions, which keeps decreasing as generations ensue. Because the language progressively becomes more and more solid and wide-spread among the 2/3 of the population who can actually communicate successfully, there is less need to adopt new constructions. This task is eventually left to the young agents only, who need to learn from the elder and the adults.

We also observed the former reduction of adoptions in the previous transmission experiment, but now the change is more noticeable, and the number of adoptions drops from 56 in the third generation to 25 in the last one. This is explained because of the increase in the population, which allows more speaker-hearer combinations and thus contributes to sky-rocketing the number of adoptions in the first few generations. We can see that in fact, the number of adoptions in the last generation displayed (with a partially consolidated language) is more comparable to the one in the experiment with a population of 10 agents.

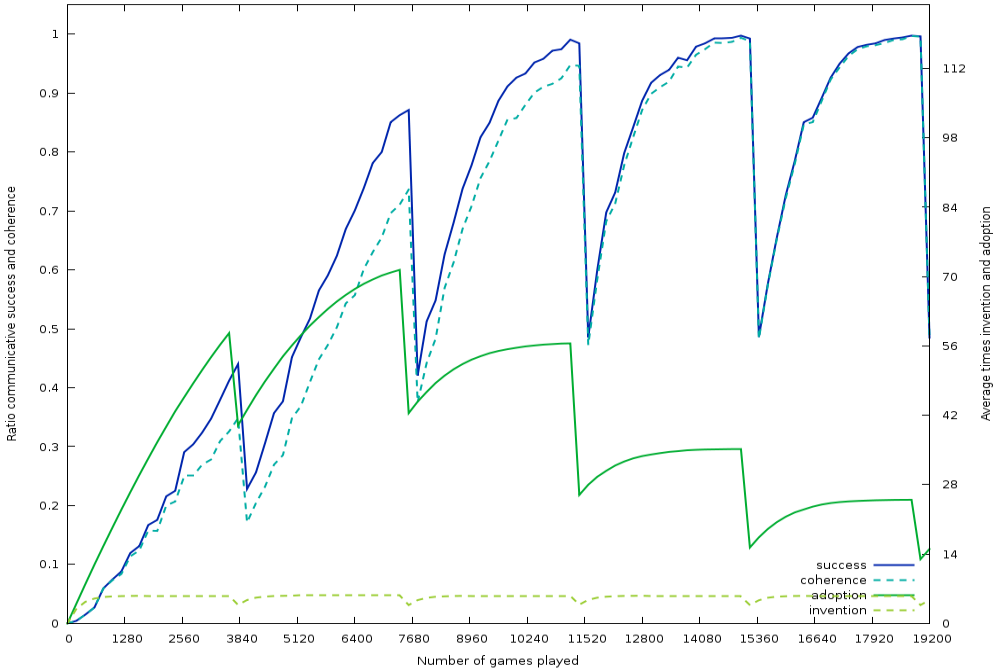


Figure 6.4: Language system transmission experiment with a population of 50 agents playing 19200 language games. Generation shifts occur every 3800 language games. The results are averaged over 10 executions and using a step of 200 games.

7. Planning

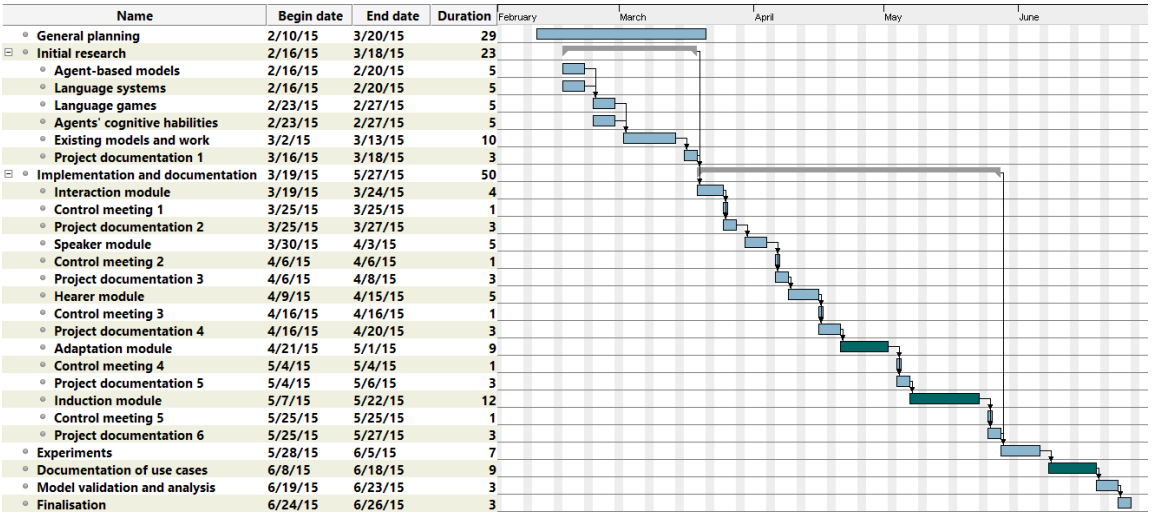
This project has been developed during the Spring semester of the academic year 2014-2015. More precisely, it began on the 14$^{\text{th}}$ of February and will finish on the 24$^{\text{th}}$ of June, when the submission deadline is set. The current chapter briefly explains the initial planning build during the initial and mandatory Project Management Course (GEP) and analyses the deviations encountered along the process.

7.1. Project Management Course (GEP)

According to the university regulations, this document must be submitted a week before the date of the defense, which in our case is the 1$^{\text{st}}$ of July. Those regulations also define a set of deadlines which were met successfully:

- 20^{th} February 2015: Deliverable 1 (scope of the project).
- 25^{th} February 2015: Deliverable 2 (temporal planning).
- 3^{rd} March 2015: Deliverable 3 (budget and sustainability).
- 8^{th} March 2015: Deliverable 4 (initial presentation).
- 15^{th} Match 2015: Deliverable 5 (contextualisation and bibliography).
- 22^{nd} March 2015: Deliverable 6 (computing specific module).
- 22^{nd} March 2015: Deliverable 7 (presentation and final document).
- 28^{th} April 2015: Progress meeting and report.
- 22^{nd} June - 27^{th} June 2015: Delivery of the final document.
- 1^{st} July 2015: Defense.

Our original schedule (previous to the start of development) and the different tasks in which the project was divided by then are represented in the Gantt chart shown below. The estimated duration for each task is also stated in the diagram. Assuming that 4.5 hours are dedicated to the project per day, the total amount of hours after combining all the tasks is of 550. This number makes it feasible to complete the project, given the appropriate amount of dedication. Note that a darker task colour in the Gantt chart denotes a task which is more difficult to complete than the rest.



Name	Begin date	End date	Duration
General planning	2/10/15	3/20/15	29
Initial research	2/16/15	3/18/15	23
Agent-based models	2/16/15	2/20/15	5
Language systems	2/16/15	2/20/15	5
Language games	2/23/15	2/27/15	5
Agents' cognitive habilities	2/23/15	2/27/15	5
Existing models and work	3/2/15	3/13/15	10
Project documentation 1	3/16/15	3/18/15	3
Implementation and documentation	3/19/15	5/27/15	50
Interaction module	3/19/15	3/24/15	4
Control meeting 1	3/25/15	3/25/15	1
Project documentation 2	3/25/15	3/27/15	3
Speaker module	3/30/15	4/3/15	5
Control meeting 2	4/6/15	4/6/15	1
Project documentation 3	4/6/15	4/8/15	3
Hearer module	4/9/15	4/15/15	5
Control meeting 3	4/16/15	4/16/15	1
Project documentation 4	4/16/15	4/20/15	3
Adaptation module	4/21/15	5/1/15	9
Control meeting 4	5/4/15	5/4/15	1
Project documentation 5	5/4/15	5/6/15	3
Induction module	5/7/15	5/22/15	12
Control meeting 5	5/25/15	5/25/15	1
Project documentation 6	5/25/15	5/27/15	3
Experiments	5/28/15	6/5/15	7
Documentation of use cases	6/8/15	6/18/15	9
Model validation and analysis	6/19/15	6/23/15	3
Finalisation	6/24/15	6/26/15	3

Figure 7.1: Gantt chart of initial tasks within the project and their durations and dependencies.

7.2. Deviations and corrections

The initial planning of this work, shown in the previous section and derived from the GEP course, was general to a certain extend, owing to the research nature of this project. At the time of planning, some details about the implementation or the most appropriate course of action to solve possible problems were not well defined or known. Similarly, the experiments we would perform after successfully implementing the model were not established.

Despite this imprecision, we have accomplished the original planning to a great extend, but with some alterations. During the development of the project minor difficulties have risen, and these have introduced some delay in the overall process and forced us to reconsider certain aspects. The initial research and familiarisation with the model took longer than expected, which caused delays in the successive implementation of each one of the modules. This implementation also took a bit longer than originally estimated to complete.

A scenario like the former was already foreseen and accounted for. Because of this, we were able to simplify the implementation of some parts of the project, still achieving functionality and maintaining the original objectives. In particular, we used a simpler induction module which does not feature chunking for rule simplification. Also we cut the number of experiments planned from three to two, covering only language emergence and transmission. Still, with these we are providing good examples of how to use and customise the modules while obtaining relevant results.

In economic terms there is not a significant change either. Unforeseen costs were calculated in case more hours were needed for unavoidable complications. However, given the simplifications we introduced in the model and the pruning of one of the planned experiments, the costs remain stable.

8. Budget and sustainability

This last chapter covers the budget of the current project and its sustainability, taking into account the scope and planning previously defined. First of all, we find a description of the costs, both material and human. It follows an analysis of possible deviations and their potential affectation. Finally, the project is viewed from the economical, social and environmental perspectives.

8.1. Budget estimation

We will now consider the required budget in order to complete the project. Expenses are divided in three different sections, depending on the nature of the resources considered: hardware, software and human. At the end, all sections are combined in a comprehensive table covering the global cost of the project. It is worth noticing that the depreciations presented are computed based on the estimated lifetime for each product and the assumption that the project is going to last 4 months.

8.1.1. Hardware resources

Hardware resources are described in Table 2. The project itself can be completed using only one computer, and no other machinery is required. This computer can be an average model with no particular requirements and will be used in all tasks, from the initial research and planning to the final presentation and document. Therefore hardware costs are considered general.

Product	Price	Estimated lifetime	Time used	Depreciation
Laptop (with peripherals)	€1000.00	4 years	4 months	€84.00
Total	€1000.00	-	-	€84.00

Table 2: Hardware budget components.

8.1.2. Software resources

Table 3 shown below summarises the costs of the necessary software to develop our project. The VirtualBox software and the Debian operating system will mainly be used during development and testing, and the other tools will be used in all tasks. Software costs are also regarded as general costs.

Product	Price	Estimated lifetime	Time used	Depreciation
Microsoft Windows 8.1	€120.00	4 years	4 months	€10.00
Debian 6.0	Free	4 years	3 months	-
Oracle VM VirtualBox 4.3.28	Free	4 years	3 months	-
Ciao Prolog 1.10#8	Free	4 years	4 months	-
Emacs 23.4	Free	4 years	4 months	-
LaTeX(TexStudio 2.8.8)	Free	4 years	4 months	-
Dropbox	Free	4 years	4 months	-
Total	€120	-	-	€10.00

Table 3: Software budget components.

8.1.3. Human resources

This project could be executed by 3 individuals with differentiated roles. Table 4 describes the cost of the human resources needed in this case. Because human resources are considered direct costs, Table 5 summarises the distribution of the expenses among the project tasks given in figure 7.1.

Role	Estimated time	Amount per hour	Cost
Project manager (M)	130 h	€40	€5200.00
Computer Science researcher (R)	356 h	€25	€8900.00
Model tester (T)	72 h	€20	€1440.00
Total	558	€28.33	€15540.00

Table 4: Human resources budget.

As it can be observed, the project manager will be responsible for general planning, supervising and resolving any logistical issues that may arise. However, the development process will be completed entirely by the researcher, who needs to implement the model itself. Finally, the tester is the person who will perform the final tests and validate the model. The latter will also take part in the development process, as a good amount of the tests need to be performed after the completion of certain modules. Therefore, because of the iterative and incremental developing methodology used in this project, the researcher and the tester need to work together.

Task	Days	Hours per day	Resource	Cost
General planning	29	4.5	M	€5200.00
Initial research	23	4.5	R	€2587.50
Implementation and documentation	50	4.5	R, T	€5400.00
Experiments	7	4.5	R,T	€755.00
Documentation of use cases	9	4.5	R	€1012.50
Model validation and analysis	3	4.5	T	€270.00
Finalisation	3	4.5	R,T	€303.75

Table 5: Human resources budget split among tasks in figure 7.1.

8.1.4. Indirect costs

We also need to consider some indirect cost derived from using the previous resources and from tools such as internet or printers. Those are summarised in Table 6.

Component	Cost
Electricity (0.1 kW / 600 hours)	€15.00
Internet	€40.00
Paper and printing	€20.00
Total	€75.00

Table 6: Summary of indirect costs.

8.1.5. Unforeseen costs

It is necessary to account for some non-predicted circumstances that may cause the project to be more expensive than previously planned as well. For example, it may be possible that more working hours are required in order to complete the project because the development did not proceed as expected. Another possibility is that our computer becomes unusable at some point during the project due to a software or hardware error. Table 7 summaries those costs and the reserved amount considered, which is proportional to the probability of an unforeseen circumstance occurring.

Component	Risk	Cost	Amount reserved
Laptop crash	0.05	€800.00	€40.00
Extended project time	0.25	€4000.00	€1000.00
Total			€1040.00

Table 7: Summary of unforeseen costs.

8.1.6. Cumulative budget

Using data from the previous tables, this section summarises all costs within the project. Here, some additional contingency amount has been added for impediments, amounting to a 10% of the total budget. The results are displayed in Table 8 below.

Component	Cost
Hardware resources	€84.00
Software resources	€10.00
Human resources	€15540.00
Indirect costs	€75.00
Contingency item (10%)	€1572.10
Unforeseen costs	€1040.00
Total	€18321.10

Table 8: Final complete budget.

8.2. Budget control

It is possible that the development of the project does not fit the a priori established plan. This is due to the limited amount of time and the complexity associated to the model. Nevertheless, in the event of any difficulty, it is certain that we will not need any additional software or hardware resources, as only technical difficulties can arise. Also, if there were any problems with the software before mentioned, we could use a free alternative available on the market. To sum up, the hardware and software budgets will most likely remain stable.

The development process is divided in various modules and takes a great portion of the total time, involving the researcher and the tester in it. We need to take into account that if the complexity of the project scales, we may require them to work more hours and consequently, to be paid more. Hiring additional workers will not help in this situation, because it is very complicated for a newcomer to make useful contributions to a partially finished model. The cost that such a circumstance may incur has already been considered as an unforeseen cost in our estimations.

To keep track of the discrepancies between the real cost of the project and the cost here computed, we will maintain the real amount expended for each one of the completed tasks in a file. At the end of a task then, we will be able to compute the deviation between the real and the hypothetical cost. Using this strategy we can quickly detect those deviations, identify where and why they occurred and calculate the additional cost necessary.

8.3. Sustainability of the project

8.3.1. Economic sustainability

This is a research-oriented project and its aim is not to build a product that may be directly purchased by customers. While in the future it may help to develop or improve other systems, at the moment is not integrated to any bigger project.

The salaries of the employees comprise nearly all the budget established for this project. In contrast, the hardware and software is cheap and constitutes a very small portion of the total cost. The total amount is very reduced and adapted to a project of this size and character.

Finally, I would like to remark that it is not possible to replicate this project in significantly less time, unless code of a previous similar implementation is recycled. In the same fashion, the whole model implemented, or parts of it, may be included in other programs or products.

We will regard this project as completely viable from the economic point of view, for its price is very affordable and it does not require using expensive equipment or working in large teams.

8.3.2. Social sustainability

With this project, we want to investigate human communicative processes and language evolution. Even though the topic is closely related to humans, the project does not directly provide any immediate benefit to the society. Similarly, and because no product is built, there is not any improvement in the consumer's quality of life or any harm to a particular individual or collective.

In the future, some other investigations or ideas may use the result of this project to aid them in developing new products that may actually be useful. This is, in part, one of the objectives of conducting this research.

This project however, does not contribute to enhancing the quality of life of any member of the society nor user. The model implemented does not directly nor necessarily unfold any enhancement in this regard. Nevertheless, it provides a tool for other researchers to further investigate in the area of language origin and evolution, which may eventually report some benefits.

8.3.3. Environmental sustainability

It is within general understanding that, after completing the project, a computer is a product that can be used for a broad range of tasks, deprecating its own cost. Since the other resources needed are mainly software, the only non environmentally friendly resource is the electricity needed to run the computer. However, even this is not an important amount compared to what a typical student would use.

We can compute an estimation of the CO_2 produced based on electricity consumption. According to governmental sources, the energy mix in Spain is approximately 267g of CO_2 per kWh. Using this data, and assuming a computer with 100W of power working an average of 5 hours per day, the carbon footprint of this project is around 15kg of CO_2.

For the aforementioned reasons, we consider this project to be very low resource-consuming, since it only requires software products and a simple computer, and the electricity consumption and carbon emissions are kept in a low level. The materials employed in building the computer and manufacturing the software are certainly complicated to estimate, but at any rate, not likely to be particularly harmful.

9. Conclusions

In this work we have successfully implemented an existing model [SSn14] to study the emerge and evolution of a language system of logical constructions. Following the original planning, this model has been tested through several experiments which simulate a scenario where a group of autonomous agents try to communicate about subsets of objects characterised by logical combinations of common basic categories.

The results obtained show that it is indeed possible for a shared vocabulary over boolean connectives and a set of grammatical constructions to emerge as the result of self-organisation processes within the population. Via the agents' interactions, each individual is able to adapt its preferences for vocabulary and grammatical constructions to those they observe are used more often by other agents.

Additionally, the experiments conducted show that the same adaptation, adoption, invention and induction mechanisms that allow a group of agents to construct a shared language system of logical constructions also enable the transmission of this very language system from one generation to the next.

Prolog has been the programming language of choice to develop our model, which is also structured in a series of reusable and independent modules implementing different functionalities within the simulation, such as the adaptation or induction mechanisms or the type of language game used. These modules can be combined in order to build models studying other kinds of language systems or using different language strategies.

The former design decisions and choice of programming language have been adopted because we would like this implementation to become a tool that can be used by other researchers in linguistics or related fields. To this effect, we also provide a detailed specification and documentation of each module that can be consulted in order to gain deeper knowledge of how the model here implemented works.

To conclude, and looking at this project from an academic point of view, I have gained insight and maturity in different topics, ranging from the working principles of agent-based models to advanced topics in logic programming and Prolog (a language in which I did not have much experience beforehand) or research results on the evolution of grammatical systems from the perspective of evolutionary linguistics. I find it valuable that computing as a field is able to open to other areas of knowledge and cooperate with other professionals in order to help develop new systems and unravel the unknown.

10. Future work

Different aspects could be extended or incorporated to the implementation provided in order to increase its functionalities or usability. The modular design of our model makes it easier to include variations in the simulation, replacing certain parts and reusing the ones which are common for all experiments. Those issues were not addressed during development because of the inherent time limitations of this project, but nevertheless some possible future upgrades are mentioned below:

- ***Spatial distribution***: Experiments studying spatial distribution [dLA00] consider the fact that some agents tend to communicate more frequently with a certain group of agents than with the rest. It is certainly a phenomenon present in real life (usually promoted by geographical features or cultural differences) that leads to the emergence and transmission of various languages within the same population. The model here developed could be used to simulate and study this as well, by altering the predicate `players` in the module `population` so that the pairs of agents chosen to participate in language games are not uniformly distributed.

- ***Guessing game***: Other types of language games could also be included in this model in order to simulate more aspects of language evolution. In particular the guessing game [Ste98], in which the hearer does not receive direct feedback on the meaning that the speaker is trying to communicate, allows studying the co-evolution of syntax and semantics. In the guessing game the speaker points to the object it wants to communicate and the hearer must construct its own conceptualization of the topic which might be different from the meaning used by the speaker. This type of language game could also be added to the model developed, by introducing variations in the modules `language_game` and `interpretation`.

- ***Graphical interface***: At the current state, the model developed needs to be compiled and executed from a terminal window passing the necessary parameters to run the simulation. While this can be done in a single command line using a provided script, it can prove difficult to some potential users not familiar with this kind of environment. A graphical interface would certainly make the program more usable and intuitive.

Bibliography

[BCC+06] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López, and Puebla G. *The Ciao Prolog System. REFERENCE MANUAL.* Technical University of Madrid, University of New Mexico, 2006.

[Bra90] I. Bratko. *PROLOG Programming for Artificial Intelligence.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1990.

[BS13] K. Beuls and L. Steels. Agent-Based Models of Strategies for the Emergence and Evolution of Grammatical Agreement. *PLoS ONE*, 8(3):e58960, 2013.

[CKPR72] A. Colmerauer, H. Kanoui, R. Pasero, and P. Roussel. In *Un système de communication en français*, Technical Report I. Groupe Intelligence Artificielle, Université Aix-Marseille II, 1972.

[CM03] William F. Clocksin and Christopher S. Mellish. *Programming in Prolog: Using the ISO Standard, Fifth Edition.* Springer-Verlag, Berlin, 2003.

[dLA00] J. de Lara and M. Alfonseca. Some strategies for the simulation of vocabulary agreement in multi-agent communities. *Journal of Artificial Societies and Social Simulation*, 3(4), 2000.

[DSB11] A. M. Di Sciullo and C. Boeckx. *The Biolinguistic Enterprise: New Perspectives on the Evolution and Nature of the Human Language Faculty.* Oxford Studies in Biolinguistics. Oxford University Press, 2011.

[GSB12] K. Gerasymova, M. Spranger, and K. Beuls. A Language Strategy for Aspect: Encoding Aktionsarten through Morphology. In *Experiments in Cultural Language Evolution*, volume 3 of *Advances in Interaction Studies*, pages 257–276. John Benjamins, Amsterdam, 2012.

[Kir02] S. Kirby. Learning, Bottlenecks and the Evolution of Recursive Syntax. In *Linguistic Evolution through Language Acquisition: Formal and Computational Models.* Cambridge University Press, 2002.

[PH12] S. Pauw and J. Hilferty. The emergence of quantifiers. In *Experiments in Cultural Language Evolution.* John Benjamins, 2012.

[SB05] L. Steels and T. Belpaeme. Coordinating perceptually grounded categories through language: A case study for colour. *Behavioral and brain sciences*, 28(4):469–529, 2005.

[SBK03] K. Smith, H. Brighton, and S. Kirby. Complex systems in language evolution: the cultural emergence of compositional structure. *Advances in Complex Systems*, 6(4):537–558, 2003.

[SS94] L. Sterling and E. Shapiro. *The Art of Prolog (2nd Edition): Advanced Programming Techniques*. MIT Press, Cambridge, MA, USA, 1994.

[SS00] J. Sierra-Santibáñez. *Revised notes of "Advanced Topics in Artificial Intelligence"*. Computer Science Department, Escuela Politécnica Superior, Autonomous University of Madrid, 2000. Chapter I: Introduction to Prolog.

[SS07] J. Sierra and J. Santibáñez. The acquisition of linguistic competence for communicating propositional logic sentences. In *Engineering Societies in the Agents World VIII, 8th International Workshop, ESAW*, pages 175–192, 2007.

[SS12] M. Spranger and L. Steels. Emergent functional grammar for space. In *Experiments in Cultural Language Evolution*. John Benjamins, 2012.

[SSn14] J. Sierra-Santibáñez. An agent-based model studying the acquisition of a language system of logical constructions. In *Proceedings of the Twenty-Eighth AAAI Conference on Artificial Intelligence, AAAI-2014*, pages 350–357. AAAI Press, 2014.

[SSn15] J. Sierra-Santibáñez. An agent-based model of the emergence and transmission of a language system for the expression of logical combinations. In *Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence, AAAI-2015*, pages 492–499. AAAI Press, 2015.

[Ste95] L. Steels. A self-organizing spatial vocabulary. *Artificial Life*, 2:319–332, 1995.

[Ste98] L. Steels. The Origins of Ontologies and Communication Conventions in Multi-Agent Systems. *Autonomous Agents and Multi-Agent Systems*, 1(2):169–194, 1998.

[Ste99] L. Steels. *The Talking Heads Experiment*. Laboratorium, Antwerpen, Belgium, 1999.

[Ste11a] L. Steels. *Design patterns in Fluid Construction Grammar*. John Benjamins, 2011.

[Ste11b] L. Steels. Modeling the cultural evolution of language. *Physics of Life Reviews*, 8(4):339–356, 2011.

[Ste12] L. Steels. Self-organization and selection in cultural language evolution. In *Experiments in Cultural Language Evolution*, pages 1–37. John Benjamins, Amsterdam, 2012.

[Vog05] P. Vogt. The emergence of compositional structures in perceptually grounded language games. *Artificial Intelligence*, 167(1-2):206–242, 2005.

[vT12] R. van Trijp. The evolution of case systems for marking event structure. In *Experiments in Cultural Language Evolution*. John Benjamins, Amsterdam, 2012.

A. User manual

The simplest way to use the program provided is by invoking the script `run.sh`. To do so please follow the next steps:

1. Open a new terminal.
2. Access the root directory of the project.
3. Run the command `./run.sh [runs] [mode] [iterations] [step] [agents]`

The `run.sh` script takes the 5 arguments detailed next:

- `[runs]`: number of complete simulations to be performed.
- `[mode]`: "emerge" for emergence experiments or "trans" for transmission experiments.
- `[iterations]`: number of language games per simulation.
- `[step]`: interval to use when collecting and producing statistics.
- `[agents]`: size of the population in each simulation.

After each run is completed, all agents will write `[end of simulation]` to its output terminal. When the overall simulation is finished, two new directories will be created:

- `./run/X/`: contains separate information of each run `X`. Grammars of each agent appear in the files `gram_Y.txt`, and `evol_com.txt` collects information about communicative success and coherence of the population. Files with the names `ad_N.txt` and `inv_N.txt` keep track of adoption and invention for agent `N`.
- `./plot/`: contains plots which represent the overall process. The mean communicate success and coherence of the population is plotted, together with the average number of adoptions and inventions. Files are generated in `png` format only.

Note: Advanced users may wish to use the Makefile provided with the following targets:

- `make all`: clean & compile targets.
- `make clean`: deletes ALL files produced by previous simulations.
- `make compile`: compiles the source code using CIAO.

B. Applicable laws and regulations

The model presented in this project is programmed using ISO-Prolog, which is affected by the standards ISO/IEC 13211-1 (core elements of Prolog) and ISO/IEC 13211-2 (support for modules in Prolog).

Additionally all the software used is free, as stated by the licenses given below:

- *GNU Emacs 23.4* : GNU General Public License (GPLv3).
  Available from `http://www.gnu.org/software/emacs/`.
- *Oracle VM VirtualBox 4.3.28* : GNU General Public License (GPLv3).
  Available from `https://www.virtualbox.org/`.
- *Debian 6.0 Squeeze* : GNU General Public License (GPLv3).
  Available from `https://www.debian.org/`.
- *Ciao System 1.10* : GNU Lesser General Public License (LGPLv2).
  Available from `http://ciao-lang.org/`

We are studying whether to offer the tool developed as free software as well, possibly under the GNU General Public Licence (GPLv3) or a similar license.