# Babel: A testbed for research in origins of language

**Angus McIntyre**
Sony CSL Paris
6 rue Amyot
Paris 75003, France
angus@csl.sony.fr

## Abstract

We believe that language is a complex adaptive system that emerges from adaptive interactions between language users and continues to evolve and adapt through repeated interactions. Our research looks at the mechanisms and processes involved in such emergence and adaptation. To provide a basis for our computer simulations, we have implemented an open-ended, extensible testbed called Babel which allows rapid construction of experiments and flexible visualization of results.

## 1 Introduction

Over the past few years, a growing number of researchers have begun to look at some of the fundamental questions in linguistics in a new light, using new tools and methodologies to explore a number of unresolved issues. Among these issues are questions about the origin and the evolution of natural languages - how a language can arise, and how it can continue to develop and change over time (see (Steels, 1997) for a summary).

Some workers in the field stick relatively closely to what might be described as the Chomskyan orthodoxy (see (Chomsky, 1981), (Chomsky, 1986)) in assuming the existence of a genetically-encoded language acquisition device (LAD) which is primarily responsible for determining the properties of language. For these researchers (see for example (Briscoe, 1997)), computer simulations offer the chance to explore the possible properties and origins of the LAD.

Other researchers choose to focus not on genetic evolution of human linguistic faculties, but on the selectionist forces that operate on language itself. Kirby and Hurford (Kirby and Hurford, 1997), for example, have shown that a model of selectionist processes operating on the language is able to explain both linguistic universals and variational constraints. The role of selection effects on language can even be explored independently of any assumed inherited language faculty; Oliphant (Oliphant, 1996) shows that communication may emerge from the nature of structured and repeated interactions between language users, while Steels (Steels, 1996) demonstrates how a coherent shared language can evolve in a population of agents as a result of repeated *language games* – stylised interactions involving the exchange of linguistic information.

Our research views language as a complex adaptive system that emerges as a result of interactions between language users. Continued adaptive interactions lead naturally to the evolution of the language and the diffusion of new linguistic tokens and properties through the community of speakers. Using computer simulations of populations of language users, we are investigating the processes that shape natural language and exploring possible learning mechanisms that can allow coherent shared communication systems to arise in populations.

This paper describes a tool that we have developed to allow rapid implementation of experimental simulations within this paradigm. Our description begins with an overview of the principal requirements we aimed to meet, followed by a more detailed look at the actual implementation of the tool and the facilities that it provides.

## 2 Requirements

Our approach to studying language is based on multi-agent simulations. Mainstream research on multi-agent systems has given rise to a number of environments and programming

languages for building simulations (consider, for example, SWARM (Minar et al., 1996), GAEA (Nakashima et al., 1996), or AKL (Carlson et al., 1994)), but none of these systems have been designed for specifically linguistic experimentation. Moreover, we wanted to work within the paradigm proposed by Steels (Steels, 1996), where language-using agents construct a shared language through repeated interactions with a precise structure. Examples of such games include *naming games*, in which agents take turns naming and learning the names of objects in their simulated environment, *imitation games* in which one agent attempts to meaningfully imitate a linguistic form presented by another, and *discrimination games*, in which agents attempt to build a system that allows them to discern distinctions between objects in the environment. The tool needed to provide a library of reusable building blocks with which we could describe the formal structure of these games, represent the principal elements of the simulated environment, and develop models of the agents' memories and learning processes. Moreover, it was important that it should be open-ended, so that we would be able to use pre-defined elements to rapidly build new simulations based on new game types or agent properties.

In addition to providing building blocks for simulation development, the system must offer an interface for controlling the simulations. This interface should allow users to launch simulations, to modify the environment by adding or removing agents, to change experimental parameters and so forth. To simplify the task of porting the tool and to protect simulation developers from the intricacies of user interface programming, we also wanted to isolate the interface code as much as possible from the code defining the (portable) core of the system and from code written by experimenters.

Lastly, the tool was required to provide ways in which the data generated by simulations could be visualized. One of the challenges in this type of simulation, particularly where multiple agents are involved, is in getting an impression of the changes that are taking place. We wanted something that could let us 'look inside' our simulations as they ran and try to get an idea of what was actually happening. It should also, of course, provide the means to export the data for subsequent analysis or presentation.

In summary, the system needed to offer an extensible set of building blocks for simulation development, tools for controlling the simulations, and tools for visualizing the progress of simulations. In the next section we will look at the approach taken to meeting these needs.

## 3 Implementation

The choice of language for the implementation was determined by the need for a standardized language suitable for rapid prototyping with good symbolic and list-processing capabilities. While the portability of Java was tempting, we eventually decided on Common LISP ((Steele, 1990)) with its more powerful symbol and list manipulation facilities.

Babel was developed using Macintosh Common LISP from Digitool, and has since been ported to Windows under Allegro Common LISP by colleagues at the Vrije Universiteit Brussel. The core of the system is portable Common LISP that can run on any platform, leaving only the interface to be ported to other platforms. In future, when stable implementations of the Common LISP Interface Manager (CLIM) are widely available, it may be possible to produce a single version which will run on any system. The task of porting is, however, not too onerous, since the majority of the code is contained in the portable core. Most important of all, experimenter code – definitions of agents, game types and environments – can typically run without modification on any platform. The high-level services provided by the toolkit mean that experimenters rarely need to get involved in platform-specific interface programming.

### 3.1 Class library

Building blocks for experimental development are provided by a rich class library of CLOS (Common LISP Object System) objects. Classes present in the library include

- basic agent classes
- classes for capturing information about interactions, the contexts in which they take place and the linguistic tokens exchanged
- classes representing the agent's environment ('worlds')

- data structures that can be used to implement agent memories and learning mechanisms

The two most important kinds of classes are the agent and the world classes. The agent classes define the capabilities of individual agents – the way they store information, the kind of utterances they can produce, and the mechanisms they use to learn or to build structure. Depending on the nature of the environment, agents may also have attributes such as position, age, energy state, social status, or any other property that might be relevant. The core class library provides a root class of agents, together with some specializations appropriate to given interaction types or learning models. Experimenters can use these classes as foundations for building agents to function in a specific experimental context.

While agent classes define the capabilities and properties of individual speakers in the language community, the world classes capture the properties of the world and, more importantly, the nature of interactions between the agents. In this way, procedural definitions of the different kinds of language games can be given as part of the definition of a basic world class. The experimenter can use a given language game simply by basing their experimental world on the appropriate class.

As an example, consider the following code fragment taken from the `ng-world` class:

```
(defmethod RUN-GAME ((World ng-world))
 (let*
  ((Speaker   (choose-speaker ...))
   (Hearer    (choose-hearer ...))
   (Context   (choose-context ...))
   (Utterance (compose-utterance ...))
   (Success
    (when Utterance
     (recognise-or-store ...))))
  (update-world-state ...)
  (register-interaction ...)))
```

This defines the basic form of the naming game - the choice of speaker and hearer, the choice of a context (including a topic), and the construction of an utterance by the speaker, followed by recognition of the utterance by the hearer[1]. The state of the world – including the agents' own memory structures – is then updated and the interaction is registered by the monitoring system (described later). Each of the methods called by this method can be individually overridden by subclasses, giving experimenters fine control over the procedures used to choose speakers or hearers, formulate utterances, store information and so forth.

[1] To make the code easier to read, function arguments are not shown

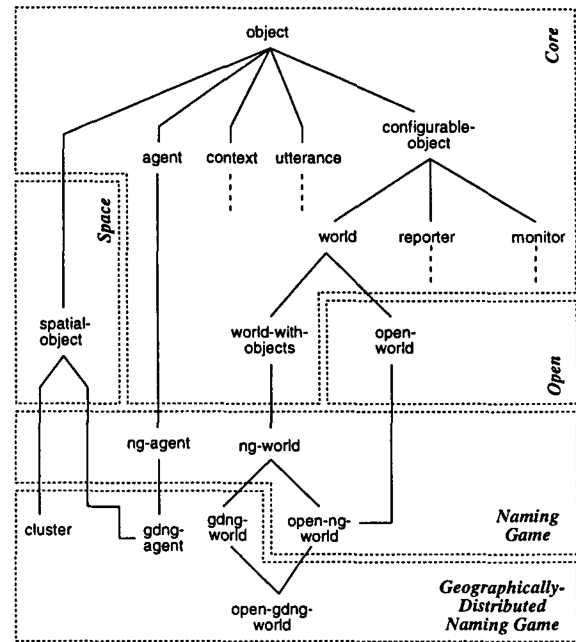


Figure 1: Core classes in Babel

The class library is implemented in a modular fashion, so that experimenters can extend the functionality of the base classes by loading additional modules. The multiple-inheritance system in CLOS allows properties to be attached to experimental objects simply by making them inherit from different subclasses. For instance, any object can be given a position by making it inherit from the class `spatial-object` defined in the **Space** module, as shown in Figure 1, which shows a portion of the existing class library.

As Babel evolves, useful classes and data structures defined by experimenters are absorbed into the core library set where they can in turn serve as building blocks for future experiments.

### 3.2 Control interface

In addition to the core class library, Babel must provide an interface that can be used to control

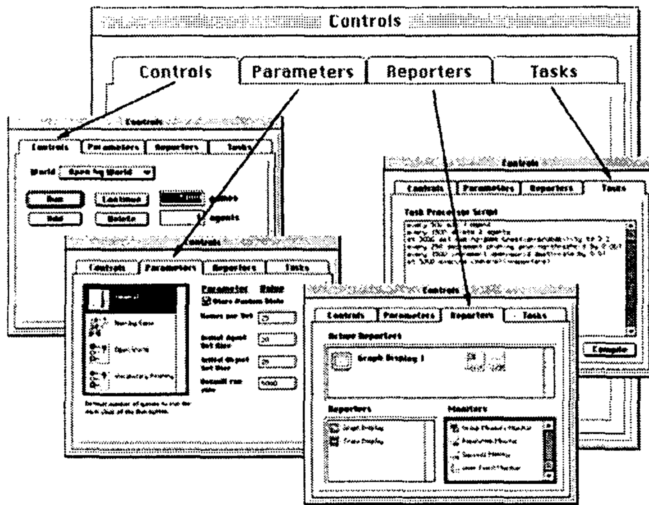


Figure 2: Babel's main control window

the simulations. As previously noted, the core Babel functions and the code defining the interface are carefully separated, in order to facilitate porting and allow experimenters to write code that does not depend on – or require knowledge of – any specific operating system platform.

The control interface in Babel is realised by a single window that allows the user to launch simulations, to set experimental parameters, to configure data reporting tools and even to write simple batch scripts to control ongoing simulations. The different functionalities are separated out into subpanes that group related controls together. Figure 2 shows a stylised view of the interface, showing each of the main control panes.

Access to interface functions is available to experimenter code through a well-defined API. For instance, experimental parameters can be declared using a simple description language that specifies the type, range and default values for each parameter. Parameters declared in this way are automatically accessible for editing through the parameter editor, and can even be updated programmatically at runtime by batch scripts executed by Babel's built-in task processor.

### 3.3 Visualization tools

A major challenge has been to provide a way to allow experimenters to follow the progress of their experiments and to view and extract data from the simulations. The same considerations that governed design of the interface are applicable here as well: the code needed to display simulation data (for instance by drawing a graph onscreen) is typically platform-dependent, but experimenters should not need to get involved in user interface programming simply to see their results. Moreover, they should not need to 'reinvent the wheel' each time; once a particular way of visualizing data has been implemented, it should be available to all experiments that can make use of a similar representation.

The approach taken in Babel has been to separate out the task of data collection from the task of data display. We call the data collectors *monitors*, because they monitor the simulation as it proceeds and sample data at appropriate intervals or under specific circumstances. Data display is handled by *reporters*, which take information from the monitors and present it to the user or export it for analysis by other programs.

Monitors and reporters stand in a many-to-many relationship to each other. The data from a given monitor type can be shown by a range of different possible reporters; in the same way, a single reporter instance can show the output from multiple monitors simultaneously. In the case of a graph display, for example, different experimental variables or measures may be drawn on the same chart, as shown in Figure 3, where change in population is graphed against communicative success over time. Similarly, a map might show the positions of individual agents and the zones of occurrence of different linguistic features. The control interface allows users to instantiate and combine monitors and reporters, while a description system allows the Babel framework to ensure that users do not attempt to combine incompatible reporters and monitors at runtime, issuing a warning if the user attempts to make an inappropriate match.

Communication between monitors and reporters is defined by a high-level API, allowing the monitors to remain platform-independent. Experimenters can build their own monitors based on a library of core monitor classes which define appropriate behaviors such as taking samples at specified intervals, reacting to events in the world or watching for the occurrence of particular conditions. Other classes may spec-

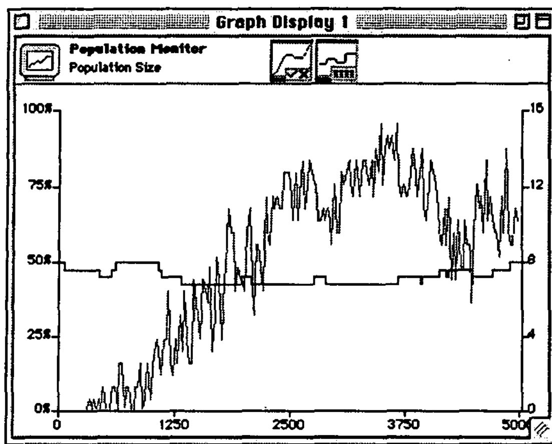


Figure 3: A graph display with two installed monitors

ify the sampling range of a given monitor – a single agent, a defined group, or the entire population – and multiple-inheritance makes it possible to flexibly combine the different types. Efforts have been made to provide powerful base classes to perform commonly-required tasks. In some cases, adding new monitoring functionality can involve as little as defining and declaring a single sampling function.

## 4 Evaluation and status

At the time of writing, the Babel toolkit is still under development, and has only been released to a very limited test group. Nevertheless, initial reactions have been generally positive, and the consensus seems to be that it meets its primary goal of simplifying and accelerating the task of developing simulations. A Windows port is in progress, and there are plans to make the software available to a wider community in future if there is sufficient interest.

## 5 Conclusion

This paper has presented an software environment for the development of multi-agent-based simulations of language emergence and evolution. Among the innovative features of the software are a class library capable of representing the stylised interactions known as language games which form the basis of our research, and a flexible mechanism for capturing and presenting data generated by the simulation.

## 6 Acknowledgements

The Babel environment was developed at the Sony Computer Science Laboratory in Paris. My colleagues Luc Steels and Frederic Kaplan of Sony CSL Paris, and Joris van Looveren and Bart de Boer from the Vrije Universiteit Brussel have provided essential feedback and suggestions throughout the development process.

## References

Ted Briscoe. 1997. Language acquisition: the bioprogram hypothesis and the baldwin effect. *Language*. (submitted).

B. Carlson, S. Janson, and S. Haridi. 1994. Akl(fd): A concurrent language for fd programming. In *Proceedings of the 1994 International Logic Programming Symposium*. MIT Press.

Noam Chomsky. 1981. *Government and Binding*. Foris, Dordrecht.

Noam Chomsky. 1986. *Knowledge of Language*. Praeger.

Simon Kirby and James Hurford. 1997. Learning, culture and evolution in the origin of linguistic constraints. In Phil Husbands and Inman Harvey, editors, *Fourth European Conference on Artificial Life*. MIT Press.

Nelson Minar, Roger Burkhart, Chris Langton, and Manor Ashkenazi. 1996. The swarm simulation system: A toolkit for building multiagent simulations. Technical report, SantaFe Institute.

Hideyuki Nakashima, Itsuki Noda, and Kenichi Handa. 1996. Organic programming language gaea for multi-agents. In Mario Tokoro, editor, *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 236–243, Menlo Park,CA. The AAAI Press.

Mike Oliphant. 1996. The dilemma of saussurean communication. *BioSystems*, 37(1-2):31–38.

Guy L. Steele. 1990. *Common LISP: The Language*. Digital Press, Bedford, MA., second edition.

Luc Steels. 1996. Self-organizing vocabularies. In C. Langton, editor, *Proceedings of Alife V*, Nara, Japan.

Luc Steels. 1997. The synthetic modeling of language origins. *Evolution of Communication Journal*, 1(1):1–34.