The Babel2 Manual

Martin Loetzsch[*], Pieter Wellens[**], Joachim De Beule[**], Joris Bleys[**] and Remi van Trijp[*]

[*]Sony Computer Science Laboratory, Paris
[**]VUB AI Lab, Vrije Universiteit Brussel

Revision 1

This document can be cited as:

M. Loetzsch, P. Wellens, J. De Beule, J. Bleys and R. van Trijp. The Babel2 manual. *AI-Memo 01-08*, AI-Lab VUB, Brussels, Belgium, 2008.

```
@TECHREPORT { babel2manual,
  author = { Loetzsch, Martin and Wellens, Pieter and
             De Beule, Joachim and Bleys, Joris and van Trijp, Remi },
  title = { The {B}abel2 Manual },
  number = { AI-Memo 01-08 },
  institution = { AI-Lab VUB },
  year = { 2008 },
  address = { Brussels, Belgium } }
```

This work was partially funded by the EU FET ECAgents Project IST-1940, by the FP7 ALEAR Project and by FWOAL328.

Contents

1 Introduction

This document serves as a technical documentation of the Babel2 framework. It is work in progress and it is our aim to continuously improve and enhance the text. This manual accompanies Babel (version 2.0) which can be downloaded for free at `http://www.emergent-languages.org`. The website also contains additional background information and downloadable papers that show in-depth experiments with Babel2 and its main component *Fluid Construction Grammar* (FCG).

Babel2 is a multi-agent framework written in common lisp developed to investigate questions concerning language evolution within the language game paradigm. Examples for experiments that were done in the framework can be found in Bleys (2008); Loetzsch et al. (2008); van Trijp (2008); Wellens et al. (2008); Steels and Wellens (2006); Steels et al. (2007); Steels and Loetzsch (2008). Babel2 consists of several modules providing specific functionality, the most important one being Fluid Construction Grammar (FCG) which provides an operational implementation of construction grammar using unify and merge operations at it's core.

Babel2 has also been designed to be totally *open-ended* with respect to the set of syntactic and semantic categories that are used by the grammar. This does not mean that the experimenter can not implement a given set of categories, but not any of the components of Babel assume any kind of universal categories. In our experiments there are typically hundreds or even thousands of new categories built. This openness of categories is in line with the Radical Construction Grammar approach which argues that linguistic categories are not universal, but construction-specific and subject to evolution (Croft, 2005).

Babel2 further subscribes to the notion of *emergent grammar* (Hopper, 1987). Languages are not seen as stable entities, but as complex adaptive systems with a great deal of variation in all levels. Several conventions may be in competition with each other, or some might not be conventionalized yet. In the FCG component this is implemented by assigning scores to constructions that are adopted based on the communicative success of the interactions that use these conventions. Moreover, the system should still come up with a reasonable parse for ungrammatical or unknown parts in an utterance, or a consistent production of new meanings and situations.

Strongly connected to that principle, Babel2 offers a usage-based model of language from a *multi-agent perspective*. The convention of constructions and their success in communication depends on the communicative interactions and needs of the language users. This emphasizes the role of the actual speakers in language change instead of treating language in its idealized form. The multi-agent aspect also means that the agents might not have completely the same language inventories and that linguistic knowledge is always local to an agent.

The primary goal of this introduction is to explain how to obtain and get Babel2 running (Section 1.1). Further it provides a general overview of what is included in the framework (Section 1.2).

1.1 Step by step instructions for getting Babel2 running

The minimal requirements to compile and run Babel2 are:

- A lisp, preferably one of: lispworks, sbcl, openmcl (section 1.1.1).
- The Babel2 source code (section 1.1.2)

The above requirements, should allow you to use the core functionality of the Babel2 framework. Babel2 provides a very powerful monitoring framework that, among other things, allows users to generate graphs (through gnuplot) and interface through a web browser. Getting these features to work is explained in sections 1.1.3 and 1.1.4. It is strongly recommended to get these working since they will greatly enhance the user experience.

1.1.1 Getting lisp up and running

If you lack a lisp installation you can either download and install lispworks (http://www.lispworks.org) or SBCL (http://www.sbcl.org). Just follow the guidelines on their respective websites. There are many sites explaining the process of setting up a lisp environment and it is outside the scope of this manual to cover this process.

1.1.2 Getting and loading Babel2 source code

The easiest way to obtain Babel2 is by downloading the latest official release from www.emergent-languages.org. When major revisions are taking place we remove the official release and you can only obtain a copy by e-mailing the authors of this document. Official releases contain all the system components and a selection of experiments and examples. Unpack the Babel2 directory to a location of your chosing. We will call this directory *Babel-home*, thus every further use of it refers to the local path to Babel on your machine.

A second way to obtain Babel2 is by using Subversion (SVN). This is only available to a limited set of users and requires password identification. The benefit is that you have access to the most up-to-date code and all of the experiments. The SVN address is http://emergent-languages.org/svn/Babel2. Please contact the authors of this document for further information of gaining access.

Assuming you have a running lisp and your Babel2 folder you can try initializing Babel2.

To initialize the Babel2 framework you have to load init-babel.lisp which resides in your Babel-home:

```
(load "Babel-home/init-babel")
```

If the above returns an error stating it cannot find asdf, you should first evaluate:

```
(load "Babel-home/libraries/asdf")
```

You should now be able to load most of the experiments in Babel-home/experiments.

You can test your installation by opening “babel-home/tests/test-babel-installation.lisp”. You should now be able to evaluate everything that is in the minimal requirements section of this file.

1.1.3 Piping to gnuplot

As explained in section 3.2.4 the monitoring system is capable of generating plots using gnuplot. For this to work it is required that at least gnuplot 4.2 is installed on your system. The monitors can also pipe output to gnuplot while running. This is only supported on UNIX operating systems (including MAC OS X and Linux). You can test these requirements by further evaluating “babel-home/tests/test-babel-installation.lisp”.

1.1.4 Interfacing through a web browser

A very nice feature of Babel2 is that for the most crucial functionality and data structures you can view these results in a web browser. For this to work you need a lisp that supports threading. During initialization Babel2 will check if your lisp supports this and if so activate a local web server, required for this to work, automatically. Lispworks supports threading on all platforms, sbcl only supports this on Linux out of the box. You can compile sbcl with support for threading on MAC OS X but this is beyond the scope of this manual.

You can verify whether a web server is running by pointing your web browser (we recommend using Firefox (any OS) or Safari (MAC OS X)) to http://localhost:8000/ and this should present you with a white page containing a small reset button. If you get a 404 error, the web server is not running.

You can also test these requirement by evaluating part 3 of “babel-home/tests/test-babel-installation.lisp”.

1.2 Overview of Babel2

The Babel2 home directory houses four important subdirectories:

- systems
- experiments
- tests
- libraries

This introduction focusses on the functionality provided by the framework which resides in `systems`. You can read more about `tests` in chapter 8. The `experiments` folder houses `single-agent-examples` which can be helpful for understanding the `fcg-core`. You can also find the `cookie-experiment` which provides an example of how to integrate the `experiment-framework` with `tasks-and-processes` and also involves some basic learning. The cookie-experiment is detailed in Chapters 6 and 5. Multi-agent examples can be found in `deconstructions`. More specifically `single-object/identfier/`

and `single-object/categorizer/multi-category/compositional/` can be of interest because they respectively provide a default implementation for a Naming Game Steels (1995) and a Guessing Game Steels and Kaplan (1999).

The most important folders (every folder here corresponds to one or more lisp packages) in the `systems` folder are (also depicted in Figure 1.1:

experiment-framework : Provides abstract classes such as `experiment`, `agent` and `world`. It also contains the base classes for learning. (see Chapter 4)

fcg-core : Module that provides the core unification and merging functionality and introduces the concepts of feature-structures, rules and rule-sets. (see Chapter 7)

tasks-and-processes : Module that provides a way to organize and run multiple interdependent smaller tasks (called processes). It provides a basic best-first search to handle ambiguity when a process returns multiple options. (see Chapter 6)

language-game-framework : Supplies instantiations of the abstract classes from the experiment-framework, such as an fcg-agent and template-experiments. It essentially glues the above three components together and thus heavily relies on them.

csc and cis : Provides constraint satisfaction functionality and a conceptual-intentional system.

monitors : A very powerful monitoring system heavily used in the framework and in the experiments. (see Chapter 3)

utils : More general utility functions and abstractions that are used throughout the framework. (see Chapter 2)

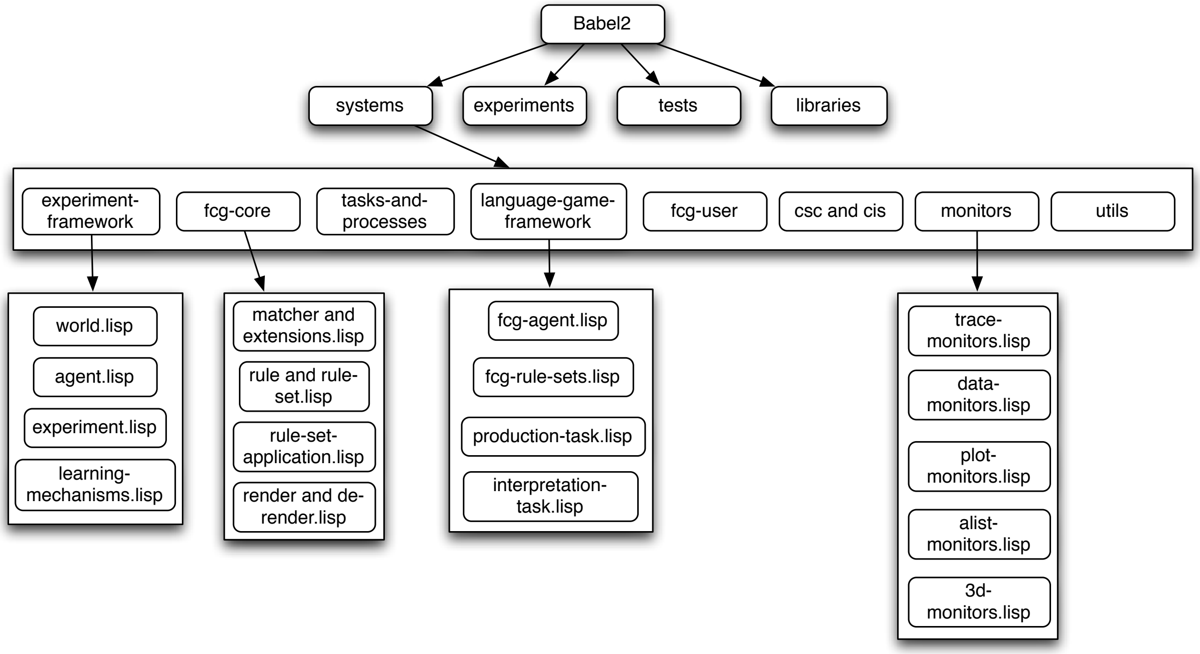

Figure 1.1: The most important folders and files found in the Babel2 framework.

2 Utilities

The *systems/utils/* folder contains a big variety of functions that were at some point considered useful in one or the other way (and many of them are frequently used). Here we will not list all of them but describe a few concepts found in the `:utils` system.

2.1 Copying Objects

Alternative hypothesis are processed in parallel. As for each alternative the agent's state will deviate from the state at the branching point, a distinct copy of the relevant state is maintained for each branch. To facilitate custom, experiment specific implementations of parts of the state, a generic copy interface is provided which should be implemented for each new data type that represents some dynamic part of the agent's state.

2.1.1 *generic function* **copy-object** *object*

description	Returns a copy of an object. Although this is the method that is called by the FCG framework to copy stuff, you will normally not implement this function but `copy-object-content` (see below) for your classes.
`object`	The thing to copy.
default implementation	There are default implementations for atomic objects such as numbers and symbols that just return the value. Strings are copied with `copy-seq`. There is a default implementation for lists that calls `copy-object` on every element of the list and collects the results. There is an implementation for `(object t)` that creates an instance of the class of `object` and calls `copy-object-content` to fill that new instance with the contents of `object`:

```
(defmethod copy-object ((object t))
  (let ((copy (make-instance (class-of object))))
    (copy-object-content object copy)
    copy))
```

2.1.2 *generic function* **copy-object-content** *source destination*

description

The function that is called by `copy-object` to do the work of copying the contents of `source` to `destination`. There are implementations for nearly all classes that are part of the FCG framework. If you add a new class and for example want to use that class as a part of a task, you will have to implement this method.

It is up to the developer to decide what is copied and what not. Normally it is assumed that `copy-object` returns a deep copy but this might often be less efficient.

`source`

The object to copy.

`destination`

The object to fill. Normally a newly created instance of the class of `source`.

example

```
(defmethod copy-object-content ((source rule-set)
                                (destination rule-set))
  (setf (slot-value destination 'type) (rule-set-type source))
  (setf (rules destination) (copy-list (rules source)))
  (setf (left-bins destination)
        (mapcar #'copy-seq (left-bins source)))
  (setf (right-bins destination)
        (mapcar #'copy-seq (right-bins source))))
```

method combination

This generic function uses the custom method combination `call-all-applicable-methods`. When `copy-object-content` is called, not only the most specific applicable method but all applicable methods are called:

```
(define-method-combination call-all-applicable-methods ()
  ((methods () :required t))
  `(progn ,@(loop for method in methods
                collect `(call-method ,method))))
```

This is very useful in such a case:

```
(defclass A () ((a :accessor a :initarg :a)))
(defclass B () ((b :accessor b :initarg :b)))
(defclass C (A B) ())

(defmethod copy-object-content ((source A) (destination A))
  (print "copy slots of class A")
  (setf (a destination) (copy-object (a source))))

(defmethod copy-object-content ((source B) (destination B))
  (print "copy slots of class B")
  (setf (b destination) (copy-object (b source))))

(copy-object object)
```

The ouput is:

```
"copy slots of class A"
"copy slots of class B"
```

The standard method combination only would have called the `copy-object-content` method for class `A` (the first super class of `C`).

2.2 Blackboards

Cognitive processes in the Babel framework share their data using a blackboard architecture.

2.2.1 *structure* **blackboard**

description	A “blackboard” is a set of “labeled” data “fields”. Such a field could be a rule set, an utterance, a meaning, etc.
slot `data-fields`	`(data-fields :type list :initform nil)` An association list consisting of `(label . value)` pairs.
copying blackboards	Blackboards are copied by calling `copy-object` (see section 2.1.1) on every field value. So you have to make sure that the `copy-object-content` method is implemented for all the classes that you store in a blackboard.

2.2.2 *generic function* **fields** *blackboard*

description/ default implementation	Returns a list containing the field labels of all fields of a blackboard.
`blackboard`	A `blackboard` instance.

2.2.3 *generic function* **field?** *blackboard label*

description/ default implementation	Whether a field exists in `blackboard`. Returns two values. The first one returns `t` only when there is a field with the given label in the blackboard. The second one returns `t` if the first one is `t` and if there is a non-nil value attached to the field.
`blackboard`	A `blackboard` instance.
`label`	A symbol that labels the field.

2.2.4 *generic function* **add-data-field** *blackboard label initial-value*

description/ default implementation	Adds a data field to a blackboard. This has to be done explicitly before a field can be read or written. Additionally, you will get an error if the field is already present in the blackboard.

`blackboard`	A `blackboard` instance.
`label`	A symbol that labels the field.
`initial-value`	A initial value for that field.
example	`(add-data-field *my-task* 'topic nil)`

2.2.5 *generic function* **get-data** *blackboard label*

description/ default implementation	Returns the value of a data field for a given label. Throws an error if the field does not exist.
`blackboard`	A `blackboard` instance.
`label`	The data field.

2.2.6 *generic function* **find-data** *blackboard label*

description/ default implementation	Returns the value of a data field for a given label. Does *not* throw an error but just returns nil if the entry is not found.
`blackboard`	A `blackboard` instance.
`label`	The data field.

2.2.7 *generic function* **set-data** *blackboard label data*

description/ default implementation	Writes some data to a data field. Throws an error if the field does not exist.
`blackboard`	A `blackboard` instance.
`label`	The data field.
`data`	A new value for that field.

2.2.8 *generic function* **remove-data** *blackboard label*

description/ default implementation	Removes both the data and the data-field from the blackboard.
`blackboard`	A `blackboard` instance.
`label`	The data field.

2.3 Trees

We provide very basic abstractions for creating and maintaining tree-like datastructures.

2.3.1 *structure* **node**

description	A node contains a parent (which is again a node) and children which is a list of nodes.
slot `parent`	`(parent nil)` Contains the parent of this node. If it is the top of the tree it contains the dummy-top-node (see below).
slot `children`	`(children nil :type list)` A list of children of this node. Of course this can be nil if it is a leaf.

2.3.2 *structure* **dummy-top-node**

description	A dummy data-type to be able to check for the root (top) of the tree. The actual top has this as parent.

2.3.3 *structure* **mtree**

description	The actual tree abstraction. Contains all of the nodes and a reference to the actual root.
slot `nodes`	`(nodes nil :type list)` A list of nodes containing all nodes of the tree.
slot `top`	`(top (make-dummy-top-node) :type node)` The actual root (or top) of the tree. Although it is initialised as dummy-top-node it should become the actual root as soon as one node is added.

2.3.4 *generic function* **has-parent?** *node*

description/ default implementation	Returns true if there is a parent which is not the dummy-top-node.
`node`	A node.

2.3.5 *generic function* **traverse** *mtree func &key from*

description/ default implementation	Traverses the mtree in a depth-first fashion calling func on every node. If from is given (which should be a node in the tree) it will not start from the top but will start from that node.
`mtree`	A tree.

`func`	A function which takes one parameter which has to be a node.
`from`	A node from which to start. If not given the top of the tree is taken.

2.3.6 *generic function* **leaf?** *node*

description/ default implementation	Returns true if the node is a leaf, which simply means it has no children.
`node`	A node.

2.3.7 *generic function* **top?** *node*

description/ default implementation	Returns true if the node is the top of a tree, which simply means it has dummy-top-node as parent.
`node`	A node.

2.3.8 *generic function* **leafs** *mtree*

description/ default implementation	Returns a list of all leafs of the tree. This is thus that subset of the nodes that have no children.
`tree`	A tree.

2.3.9 *generic function* **add-node** *mtree node &key parent*

description/ default implementation	Adds the given node to the tree. Although it can only do so as a leaf. So it cannot add a node in the middle of the tree or at the top if there already is a top. Therefore you have to supply :parent when this is not the first node added to the tree. The parent has to be valid.
`tree`	A tree.
`node`	The new node that shoud be added to the tree.
`parent`	The node in the tree to which you wish to hang the new node.

2.3.10 *generic function* **replace-node** *mtree old-node new-node*

description/ default implementation	replaces the old node with the new node. It can only replace leaf. Of course the old-node has to be found in the tree.
`tree`	A tree.
`old-node`	The old node to be replaced.
`new-node`	The node that should replace the old one.

2.3.11 *generic function* **depth** *node*

description/ default implementation	Returns the depth of the given node in the tree from which it is part. It is not required to pass the tree itself.
`node`	The node form which you wish to know its depth.

3 Monitoring Experiments

The Babel framework contains extensive monitoring and debugging mechanisms that help developers and users to

- print comprehensible traces of the execution of specific components on the screen (e.g. process execution, learning framework, games ...),
- raise warnings or take other actions when specific events happen,
- record and store arbitrary numeric and non-numeric values for each interaction,
- print these data to the screen or write them to a file
- plot these data in real-time using gnuplot or generate graphs offline.

This chapter describes the general monitoring mechanisms, the built-in monitors classes that come with the monitor system, and helps you to use monitors for your own experiments. The monitoring system is defined in directory *systems/monitors*.

3.1 Events, Monitors and Notfications

The main motivation for implementing the monitor system was to separate the source code that does something (running an experiment, running a production task, repairing something, etc) from source code for debugging and data collection. Thereto, a set of *"monitors"* that subscribe to a set of *"events"* are defined. In some source code that does something, the monitoring system is *"notified"* for a specific event. The *"active"* monitors *"handle"* that event. Here is an example:

```
(in-package :monitors)

(define-event run-test-finished (result number))

(defun run-test ()
  (let ((result (random 10)))
    ;; do something
    (notify run-test-finished result)
    result))

(define-monitor print-test-result :documentation "prints the result of run-test")

(define-event-handler (print-test-result run-test-finished)
  (format t "~%run-test finished. Result: ~a" result))
```

The function `run-test` does something, amongst other things calculating the variable `result`. Let's assume a developer wants to print the result of function `run-test` whenever it finishes. There is an event `run-test-finished` defined. At the end of function `run-test`, the monitoring system is notified on that event, passing the value of variable `result` as a parameter. Then there is the definition of monitor `print-test-result`, together with an event handler that handles the event for that monitor by printing the result. However, if you run `run-test` like it is, nothing will happen. The monitor needs to be activated:

```
(activate-monitor print-test-result)
```

Only active monitors get notified on their events. This helps you in deciding which information to print, record, plot etc. In this case, the monitor will print:

```
MONITORS> (run-test)

run-test finished. Result: 6
```

This seems to be a lot of code to just print the result of function `run-test`. However, you could easily add more monitors that handle the same event:

```
(define-monitor warn-when-test-result-is-7
    :documentation "warns when function run-test returns 7")

(define-event-handler (warn-when-test-result-is-7 run-test-finished)
  (when (= result 7) (warn "function run-test returned 7!!!!")))
```

Or there could be another monitor that plots the result of `run-test` in a graph. The benefit of using the monitor system is that you do not clutter up your functions with code that does not contribute to the computation of the function's result. Despite that, you get a lot of things "for free", as you will see in the remainder of this chapter.

The rest of this section describes the macros that define events and monitors, that notify on events, and that activate monitors. If you are interested in the classes and methods that these macros are based on, then you might want to read section 3.3 first.

3.1.1 *macro* **define-event** *id &rest parameters*

description	Defines an event. This needs to be done before any notification or handler on that event can be defined. Typically, such a definition is put directly before the definition of the function that notifies on that event.
`id`	The id for that event. A symbol.
`parameters`	The parameters with that the event is notified. These are lists with a variable name and the type of the parameter. This information is used to check that you pass the right values with a notification. In a handler you can rely on the passed parameters being of these types.

example	`(define-event game-finished (result symbol)`
	`  (speaker fcg-agent) (hearer fcg-agent))`
	There could be a notification for this event at the end of a game. The first parameter would have to be a symbol and the other two instances of (derivates of) `fcg-agent`.

3.1.2 *macro* **define-monitor** *id &key class*

description	Defines a monitor. This should happen before any event handlers for that monitor are defined. Typically, monitors and their handlers are in the same source file.
`id`	An id for the monitor. Typically starts with a verb describing the purpose of the monitor, e.g. `print-`, `trace-`, `record-`, `plot-` etc.
`:class`	The name of the monitor class. As it will be explained in the next section 3.2, there is a big variety of monitor subclasses. The macro `define-monitor` is used to instantiate all of them. If `:class` is not provided, the monitor base class `monitor` is used.
`:documentation`	This should be a short text that helps other users to guess what the monitor does.
other keyword parameters	Depending on `:class`, other keyword parameters are allowed. These are described in detail in section 3.2.

example

```
(define-monitor print-game-result
    :documentation "prints the result of a game")
```

This defines a monitor based on class `monitor` (because no `:class` parameter was provided).

3.1.3 *macro* **define-event-handler** *(monitor-id event-id) &body body*

description	Installs a method that handles a specific event for a monitor (or a list of monitors).
`monitor-id`	The id of the monitor that handles the event. Can be also a list of multiple monitor ids. The event is then handled in the same way for these monitors.
`event-id`	The event to handle.
`body`	An expression that handles the event. You can access the parameters using the variable names from the event definition. If you don't know the event parameters, use function `print-all-monitors` to get the definition. The monitor instance can be accessed using the variable `monitor` (see also the example below).

example

```
(define-event-handler (print-game-result game-finished)
  (format t "~%Game finished. Result: ~a" result))
```

In the background, this method will be created:

```
(defmethod  handle-game-finished-event
    ((monitor monitor) (monitor-id (eql 'print-game-result))
     (event-id (eql 'game-finished)) (result symbol)
     (speaker fcg-agent) (hearer fcg-agent))
  (format t "~%Game finished. result: ~a" result))
```

This basically makes a method that uses all the parameters of the event definition as method parameters.

3.1.4 *macro* **notify** *event-id &rest parameters*

description	Notifies the monitor system that an event happened. For all monitors that can handle the event and that are active, the event handler is called. The order in which the handlers are called is not specified.
`id`	The id of the event
`parameters`	Values for parameters of the event. These need to be as specified in the definition of the event. You will get an error if you pass the wrong number of parameters or if the passed values don't match the parameter types of the event definition.

example

```
(defun run-game ()
  ;; ... do something
  (notify game-finished result speaker hearer))
```

Don't hesitate to use `notify` statements in even time-critical code. As you can see below in the macro expansion, it only loops over the active monitors of an event and calls the corresponding handler methods. When there are no active monitors for that event, nothing happens. Please refer to section 3.3 for more details.

```
(dolist (monitor-id (active-monitors (get-event 'game-finished)))
  (let ((monitor (get-monitor monitor-id)))
    (handle-game-finished-event monitor monitor-id
     'game-finished result speaker hearer)))
```

3.1.5 *macro* **activate-monitor** *id &optional active*

description	Activates a monitor.
`id`	The id of the monitor.
`active`	Defaults to `t`. When `nil`, the monitor is deactivated. This can be used to write functions that activate/ deactivate a set of monitors together.

3.1.6 *macro* **deactivate-monitor** *id*

description	Deactivates a monitor.
`id`	The id of the monitor.

3.1.7 *macro* **toggle-monitor** *id*

description	Toggles (inverses) the activation of a monitor.
`id`	The id of the monitor.

3.1.8 *macro* **toggle-monitors** *&rest ids*

description	Toggles (inverses) the activation of a list of monitors.
`ids`	A list of monitor ids

3.1.9 *macro* **print-all-monitors**

description	Prints for all defined monitors their type, wether they are active, the documentation and the source file they are defined in (if possible).

3.1.10 *macro* **print-all-events**

description	Prints for all defined events the parameter list and the source file they are defined in (if possible).

3.1.11 Pre-defined Events

The monitoring system is designed to monitor experiments. Many of the built-in monitor classes do something at the end of interactions, series, or batches (see section 4.2.3). For example updating a plot, writing data to a file or printing something to the screen. They "know" that for example an interaction finished because the experiment framework notifies on these built-in events (section 4.2.3):

3.1.11.1 *monitor event* **interaction-started** *(experiment t)*

description	Notified at the beginning of an interaction.
`experiment`	The current experiment (contains the interaction number). The type in this definition is `t` is because the class `experiment` is not known yet in the context of the monitor system.

3.1.11.2 *monitor event* **interaction-finished** *(experiment t) (interaction-number fixnum)*

description	Triggered after each interaction.
`experiment`	The current experiment.
`interaction-number`	The current interaction number of the experiment.

3.1.11.3 *monitor event* **series-finished** *(series-number fixnum)*

description	At the end of a series.
`series-number`	The number of the finished series (starting from 1).

3.1.11.4 *monitor event* **batch-finished** *(experiment-class string*

description	Notifies that a batch finished.
`experiment-class`	The name of the experiment class.

3.1.11.5 *monitor event* **reset-monitors**

explanation	You can notify on this event in order to reset the monitor system.

These events are also very handy to define event handlers for your own monitors. For example with the event `interaction-finished` you get the experiment as a parameter and can record some measures of the rule-sets of the agents of your population.

3.2 Built-in Monitor Classes

You will rarely define monitors of the base class `monitor`. Instead, you would choose one of the built-in classes that provide a big variety of additional functionalities. Or one of the classes that you defined yourself (see section 3.3.3). Figure 3.2 gives an overview over the hierarchy of built-in monitor classes.

The rest of this section describes how to use them. Although the different kinds are referred to with their class name, the slots of these classes are not described here (you normally also don't get in touch with instances of monitors), as monitors of all classes have to be defined with the `define-monitor` macro (see previous section).

3.2.1 Printing Program Traces

The most common debugging technique when programming lisp is to add `format` statements to the functions and comment them out when they are not needed anymore. However, this can become quite cumbersome if you want to print different stuff in different situations. The built in *"trace*

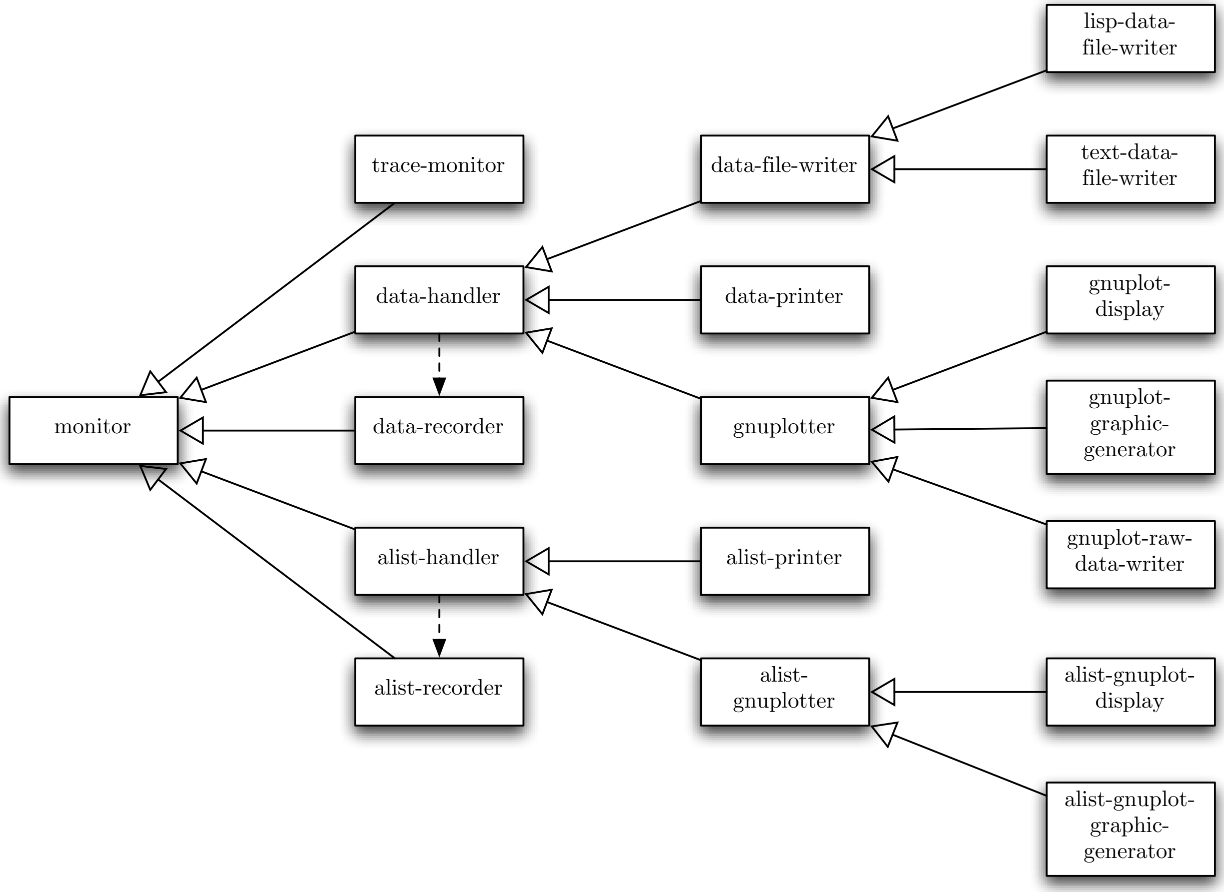

Figure 3.1: The class hierarchy of the built-in monitor classes

monitors" allow you to toggle printing of information by toggling monitor activation. Additionally, they can buffer text messages and print them later on demand.

3.2.1.1 *monitor class* **trace-monitor** *monitor*

description	A monitor for printing text messages to the screen. It is also able to buffer the text (see below).
example	`(define-monitor trace-run-test :class 'trace-monitor :documentation "Traces the execution of run-test.")`
message handlers	In message handlers, use the method `(monitor-stream monitor)` as the stream to write the text message to. This makes it possible to buffer the messages.

```
(define-event-handler (trace-run-test run-test-finished)
  (format (monitor-stream monitor)
    "~%run-test finished. Result: ~a" result))
```

passing text messages directly

It would not make sense to always define a new event when you just want to print some text somewhere. Thereto, there is also the possibility to notify trace monitors directly on text messages. When a trace monitor is created, an event with the same name as the monitor and a message handler for the monitor and that event are automatically defined. In the example above, the event `trace-run-test` with a parameter `message` of type `string`, and a message handler that puts the message on the stream of the monitor are automatically created. This makes it possible to write things like this:

```
(defun run-test ()
  (let ((result (random 10)))
    ;; do something
    (notify trace-run-test
      (format nil "~%in the middle of run-test!"))
    ;; do something
    (notify run-test-finished result)
    result))
```

After activating the monitor `trace-run-test`, the execution of run-test prints this:

```
MONITORS> (run-test)

in the middle of run-test!
run-test finished. Result: 2
2
```

3.2.1.2 *function* **activate-buffering-of-trace-monitors**

description

Switches on buffering. Messages for trace monitors are not printed immediately anymore. Instead, they are kept in a buffer (and can be printed later).

This handy feature makes it possible to get to a lot of different traces after something happened without having to print them to the screen beforehand.

example

```
MONITORS> (activate-buffering-of-trace-monitors)
NIL
MONITORS> (progn (run-test) (run-test))
NIL
MONITORS> (print-buffered-messages-of-trace-monitors)

in the middle of run-test!
run-test finished. Result: 7
in the middle of run-test!
run-test finished. Result: 9

NIL
```

(for the printing of buffered messages see below)

3.2.1.3 *function* **print-buffered-messages-of-trace-monitors**

description	Prints all the buffered messages of all trace monitors.

3.2.1.4 *function* **clear-trace-monitors-buffer**

description	Clears the buffer for the trace monitors. Does not change activation state. In order to avoid the buffer to become very large, it is recommended to call this function often, for example before each interaction.

3.2.1.5 *macro* **deactivate-buffering-of-trace-monitors**

description	Deactivates the buffering for trace monitors.

3.2.1.6 *generic function* **print-with-overline** *monitor character message*

description/ default implementation	Prints a text message in a trace monitor with an overline of the same length as the message (see example below).
`monitor`	An instance of `trace-monitor`.
`character`	The character to use for printing the overline.
`message`	The message itself. Should not contain line breaks.

example

```
(define-event-handler (trace-language-game interaction-started)
  (print-with-overline monitor #\=
       (format nil "= Started interaction ~a."
       (interaction-number experiment))))
```

The resulting output:

```
=========================
= Started interaction 28.
```

3.2.2 Recording Data

One of the purposes of running experiments is to output quantitative measures for the emergence of some feature of language, occurrences of some event, properties of the population, and so on. These measures could be plotted to a graph, written to a data file or printed to the screen. One such a measure could be for example communicative success.

To avoid that all plotting, data-writing and printing monitors which want to output communicative success have to collect that measure for themselves, there is the separation of monitors that record data (class `data-recorder`, this section) and monitors that output these recorded data (next sections 3.2.3 and 3.2.4). For example there is only one monitor that records the communicative success. All monitors that output communicative success in some way use the recorded data of that monitor.

3.2.2.1 *monitor class* **data-recorder** *monitor*

description — A monitor that records a single value for each interaction. If you run a batch of series, it also keeps the values of each series. For example after a `(run-batch my-experiment 100 3)`, the monitor will have recorded 100 × 3 values. The values are recorded from message handlers (see below). If the recorded values are numerical, then the data recorder also stores values that are averaged over a window.

example

```
(define-monitor record-success
    :class 'data-recorder :default-value 0 :average-window 100
    :documentation "records whether the game was a success")
```

`:default-value` — The default value to record. Default: 0. If the data recorder did not receive any other value during an interaction, this value will be recorded.

`:average-window` — The average values will be computed as the arithmetic mean of the last `:average-window` values. Default: 100. The computation is fast as it does a recursive update after 2 × `:average-window` interactions.

recording values — Values are recorded from message handlers. An example:

```
(define-event-handler (record-success game-finished)
  (if (eql result 'succeed)
      (record-value monitor 1)
      (record-value monitor 0)))
```

In the handler, the function `record-value` is used to get the next value into the data recorder. If this function is called multiple times within an interaction, only the last recorded value is stored.
Alternatively, you can use `incf-value` when you want to sum the values of several events during one interaction:

```
(define-event-handler (some-monitor some-event)
  (incf-value monitor some-value))
```

3.2.3 Outputting Recorded Data

All monitors that somehow output data recorded by data recorders are derived from this class:

3.2.3.1 *monitor class* **data-handler** *monitor*

description — An abstract class that provides functionality to access data of data recorders.

example

```
(m:define-monitor my-data-handler :class ’data-handler
    :data-sources ’(record-utterance
                    (average record-success)
                    record-number-of-lex-stem-rules))
```

This monitor will output the recorded values of the data recorders `record-utterance`, `record-success` and `record-number-of-lex-stem-rules`.

However, it will not make sense to define a monitor of class `data-handler`, as it does not output anything. You will rather use one of the derived classes from below or the next section.

`:data-sources`

A list of data recorder ids. These have to be defined before a data-handler on them can be defined. If you want to access the averaged values of a data recorder, you write `(average monitor-id)`, as in the example above.

activation

When you activate any monitor that is derived from `data-handler`, all the data recorders that are specified in `:data-sources` will automatically become activated.

3.2.3.2 *monitor class* **data-printer** *data-handler*

description

A monitor for printing recorded data to the screen.

example

```
(m:define-monitor my-experiment-printer :class ’data-printer
    :data-sources ’(record-utterance
                    (average record-success)
                    record-number-of-lex-stem-rules))
    :format-string "~%~d: ~30a ~,2f ~,2f"
    :interval 50)
```

This monitor prints the interaction number, the utterance, the average communicative success and the lexicon size to the screen:

```
MY-EXPERIMENT> (run-interactions 500 :reset t)

50: (fulele napimu xifibu)         0.08 13.00
100: (monalu)                       0.10 20.60
150: (sopogo)                       0.12 26.40
200: (zaradu fulele)                0.19 32.20
250: (fenowu foxapu)                0.43 35.20
300: (vazoro kesoku)                0.64 38.40
350: (wunoze)                       0.75 40.20
400: (raneni napimu)                0.81 42.20
450: (giwene xifibu)                0.83 43.40
```

`:data-sources`

See section 3.2.3.1.

`:format-string`

A format string as in `format`. The first argument to it is the interaction number. The other arguments are the current values of the data recorders a specified in `:data-sources`.

`:interval`	How often the information is printed. Default: 1.

3.2.3.3 *monitor class* **data-file-writer** *data-handler*

description	An abstract class for writing recorded data to a file. You can not define monitors of this class. See the two derived classes below instead.

3.2.3.4 *monitor class* **text-data-file-writer** *data-file-writer*

description Writes recorded data in columns to a text file, which can be imported for example in Excel. For each data source there is for each series a separate column.

example

```
(define-monitor write-success-and-lexicon-size-to-file
    :class 'text-data-file-writer
    :data-sources '((average record-success)
                    record-number-of-lex-stem-rules)
    :file-name (make-pathname :directory '(:absolute "tmp")
                              :name "success-and-lexicon"
                              :type "dat")
    :add-time-and-experiment-to-file-name t
    :column-separator " "
    :comment-string "#")
```

The text file is written at the end of a batch:

```
MY-EXPERIMENT> (run-batch 500 3)

monitor write-success-and-lexicon-size-to-file:
  wrote /tmp/2007-03-29-16-25-naming-game-success-and-lexicon.dat
NIL
```

The generated file */tmp/2007-03-29-16-25-naming-game-succes-and-lexicon.dat* looks like this:

```
# This file was created by the
# text-data-file-writer WRITE-SUCCESS-AND-LEXICON-SIZE-TO-FILE.
# The columns are:
# interaction number
# RECORD-SUCCESS-0
# RECORD-SUCCESS-1
# RECORD-SUCCESS-2
# RECORD-NUMBER-OF-LEX-STEM-RULES-0
# RECORD-NUMBER-OF-LEX-STEM-RULES-1
# RECORD-NUMBER-OF-LEX-STEM-RULES-2
0.0 0.0 0.0 0.0 0.4 0.4 0.4
1.0 0.0 0.0 0.0 0.4 0.8 0.8
2.0 0.0 0.0 0.0 0.8 1.2 1.2

...
498.0 0.9 0.92 0.88 42.8 39.6 47.2
499.0 0.9 0.92 0.88 42.8 39.6 47.2
500.0 0.9 0.92 0.89 42.8 39.6 47.2
```

`:data-sources`	See section 3.2.3.1.
`:file-name`	The name of the file to generate. Should be a `pathname`.
`add-time-and-experiment-to-file-name`	When `t` (default), the current date and time as well as the name of the experiment class are added to the file name.
`:column-separator`	The string to separate values. Default: `" "`.
`:comment-string`	How to start comment lines. Default: `"#"`.

3.2.3.5 *monitor class* **lisp-data-file-writer** *data-file-writer*

description	Writes the recorded data as a single s-expression to a file. This can be handy if you later want to read the data back to lisp.

example

```
(define-monitor write-success-and-lexicon-size-to-file
    :class 'lisp-data-file-writer
    :data-sources '((average record-success)
                     record-number-of-lex-stem-rules)
    :file-name (make-pathname :directory '(:absolute "tmp")
                              :name "success-and-lexicon"
                              :type "lisp")
    :add-time-and-experiment-to-file-name nil)
```

The resulting text file can be read back with

```
(with-open-file (stream #P"/tmp/success-and-lexicon.lisp")
    (defparameter data (read stream)))
```

`:data-sources`	See section 3.2.3.1.
`:file-name`	The name of the file to generate. Should be a `pathname`.
`add-time-and-experiment-to-file-name`	See section 3.2.3.1.

3.2.4 Plotting Data with Gnuplot

The probably most prominent feature of the monitoring system is to produce graphs from recorded data using gnuplot.

3.2.4.1 *monitor class* **gnuplotter** *data-handler*

description	An abstract class for plotting data with gnuplot. Classes that derive from this one define how the resulting graph is displayed or written.

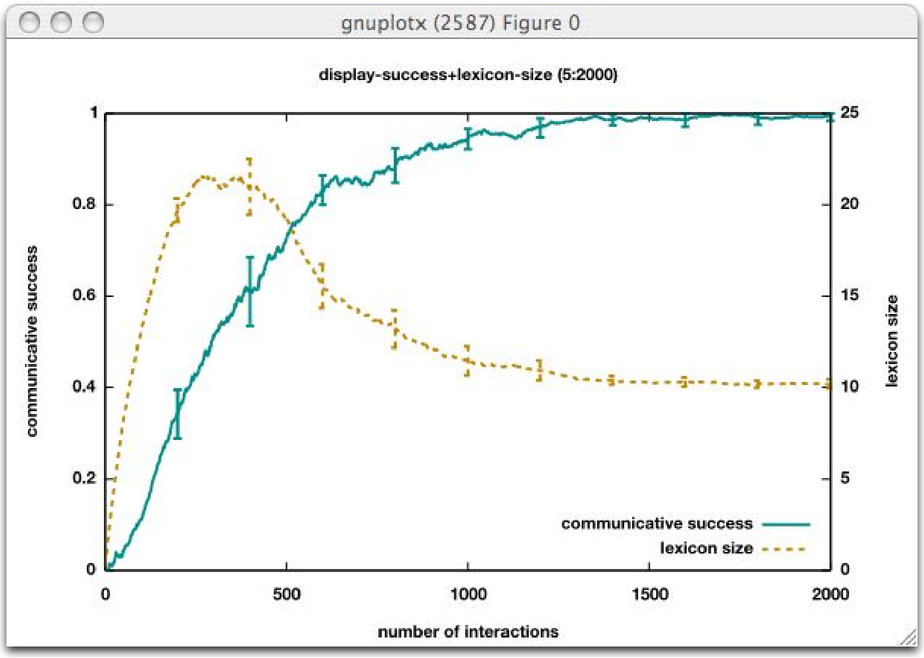

Figure 3.2: An example for a real-time plot with a `gnuplot-display` monitor

example

```
(define-monitor plot-success-and-lexicon-size
    :class 'CLASS-DERIVED-FROM-GNUPLOTTER
    :data-sources '((average record-communicative-success)
                    record-average-number-of-words)
    :caption '("communicative success" "lexicon size")
    :minimum-number-of-data-points 500
    :x-label "number of interactions"
    :y1-label "communicative success"
    :y2-label "lexicon size"
    :error-bars t :draw-y1-grid nil
    :line-width 2
    :use-y-axis '(1 2)
    :y1-max 1 :y1-min 0
    :y2-min 0 :y2-max nil
    :colors *great-gnuplot-colors*
    :key-location "right bottom")
```

Depending on what class `CLASS-DERIVED-FROM-GNUPLOTTER` is, the result might look as in figure 3.2 or figure 3.3.

`:data-sources` See section 3.2.3.1.

`:caption` Captions for each data source for generating a graph legend. Should be list of strings with one caption per data source. When no captions are provided, the names of the data recorders are used.

`:minimum-number-of-data-points`	The “sample rate”. For at least that many points on the x-axis there will be a value plotted. Smaller numbers here result in higher speed and smaller graphic files, bigger numbers give a higher resolution. Default: 500.

`:x-label`
`:y1-label`
`:y2-label`
	Labels for the x-axis, left y-axis and right y-axis. Default: `nil`.

`:error-bars`	When `t`, error bars are added to the graph. Default: `nil`.
`:draw-y1-grid` `:draw-y2-grid`	When `t`, thin horizontal lines are drawn at the ticks of the left or right axis. Default: `nil`.
`:line-width`	The width of the lines for curves and errorbars. Default: `2`.
`:use-y-axis`	A list of values 1 or 2 for each data source specifying whether to scale the data with the left y-axis (1) or the right (2). Default: scale all with the left y-axis.
`:y1-min :y1-max` `:y2-min :y2-max`	Minimum and maximum values for the left and right y-axis. When not provided, the data is automatically scaled to fit (which is often better).
`:colored`	Whether the graph uses colors for different data lines nor not (only has effect on some gnuplot terminals). Default `t`.
`:colors`	A list of colors (list of strings, e.g. `'("red", "green")`) to be used by the different graph lines. Defaults to `*great-gnuplot-colors*`.
`:key-location`	Where to put the graph legend. Possible values are for example `"off"` (for no legend) `"below"`, `"top right"`, etc. For more information, type `"help set key"` into gnuplot. Default: `"below"`.

3.2.4.2 *monitor class* **gnuplot-display** *gnuplotter*

description

Plots the data in real-time into a window using gnuplot.

This requires the proper installation of a recent version of gnuplot and a configuration of your lisp environment so that it can find gnuplot. You can test whether your lisp/gnuplot integration works by loading the `:utils` asdf system and evaluating:

```
(utils:with-open-pipe
    (stream (utils:pipe-output "gnuplot" :args '("-persist")))
  (format stream "plot sin(x)")
  (finish-output stream))
```

This should open a window that shows a sinus plot.

example

```
(define-monitor display-success-and-lexicon-size
    :class 'gnuplot-display
    :documentation "Shows communicative success and lexicon size."
    :data-sources '((average record-communicative-success)
                    record-average-number-of-words)
    :caption '("communicative success" "lexicon size")
    :x-label "number of interactions"
    :y1-label "communicative success" :y2-label "lexicon size"
    :error-bars t :use-y-axis '(1 2)
    :y1-max 1 :y1-min 0 :y2-min 0 :key-location "right bottom")
```

When activating this monitor and for example running `(run-batch 2500 5)` a plot window such as in figure 3.2 will be updated every 50 interactions.

`:update-interval` How often the graph display will be updated. The smaller the value, the slower the whole thing. Default: 10 (very slow).

All other parameters are the same as in section 3.2.4.1.

3.2.4.3 *monitor class* **gnuplot-graphic-generator** *gnuplotter*

description Produces a graph file at the end of a batch. It is recommended to use such a graph for papers instead of saving the result of `gnuplot-display-monitors`, as they produce graphic files of much higher quality.

example

```
(define-monitor plot-success-and-lexicon-size
    :class 'gnuplot-graphic-generator
    :documentation "Plots communicative success and lexicon size"
    :graphic-type "pdf"
    :colored nil
    :add-time-and-experiment-to-file-name nil
    :file-name (make-pathname :directory '(:absolute "tmp")
                              :name "success-and-lexicon-size"
                              :type "pdf")
    :data-sources '((average record-communicative-success)
      record-average-number-of-words)
    :caption '("communicative success" "lexicon size")
    :x-label "number of interactions"
    :y1-label "communicative success" :y2-label "lexicon size"
    :error-bars t :use-y-axis '(1 2) :key-location "right bottom"
    :y1-max 1 :y1-min 0 :y2-min 0 :draw-y1-grid t)
```

The resulting file */tmp/success-and-lexicon-size.pdf* looks as in figure 3.3.

`graphic-type` Which gnuplot graphic driver to use. Should be one out of `"postscript"`, `"pdf"`, `"svg"` or `"gif"`.

`:file-name` The file name of the graphic file to produce. Should be a `pathname`.

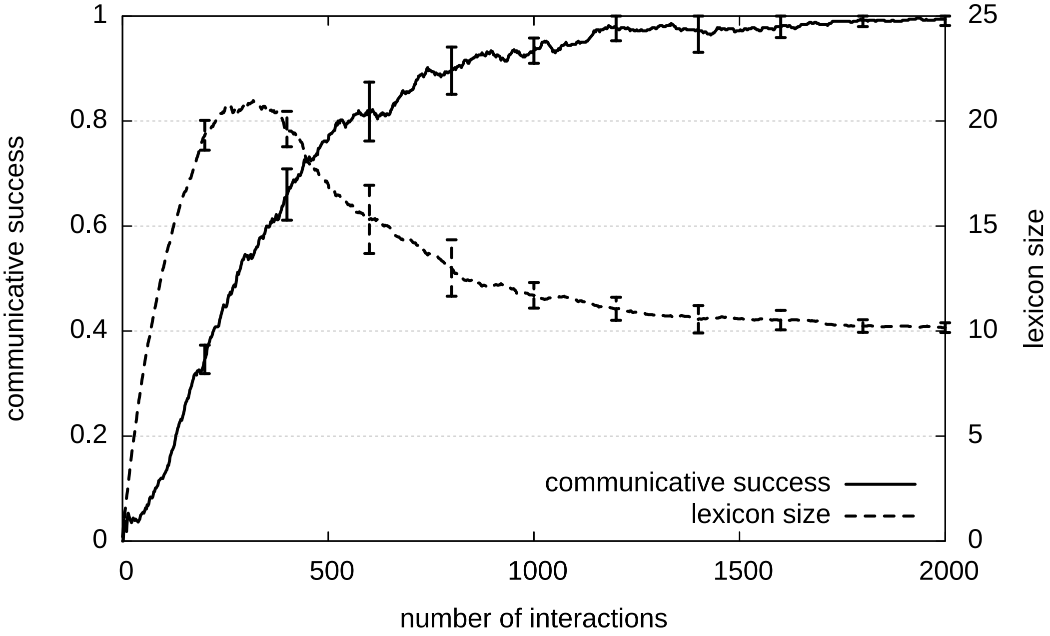

Figure 3.3: An example for a plot generated by a `gnuplot-graphic-generator` monitor

`:add-time-and-experiment-to-file-name`	See section 3.2.3.4.
	All other parameters are the same as in section 3.2.4.1.

3.2.4.4 *monitor class* **gnuplot-display-and-graphic-generator** *gnuplot-display gnuplot-graphic-generator*

description	A monitor that produces a real-time plot and generates a graphic file in the end. As it derives from both of these classes, it takes all parameters that `gnuplot-display` and `gnuplot-graphic-generator` take.

3.2.4.5 *monitor class* **gnuplot-file-writer** *gnuplot-graphic-generator*

description	If you don't like the graphs that the `gnuplot-graphic-generator` makes, you can also use this monitor to write a complete gnuplot script (with data and plot commands) and later modify that script. The parameter `:gnuplot-file-name` specifies the file name of the script.

3.2.5 Recording and Plotting Lists of Data

Data recorders (section 3.2.2.1) allow to record one single value for each interaction. But sometimes one wants to look at the evolution of a list of values, for example the scores of some entities or the number of occurrences of events. Defining a data recorder for each of them would become cumbersome, especially if the number of values is not known beforehand. For this purpose, there are monitors for recording and processing lists of data.

3.2.5.1 *monitor class* **alist-recorder** *monitor*

description

Records averaged values for lists of `(symbol . value)` conses. Similar to a `data-recorder` (section 3.2.2.1), values are kept for each interaction of each series of a batch.

example

```
(define-monitor record-lexicon-of-first-agent-for-first-object
    :class 'alist-recorder
    :documentation "Records for the first agent the scores of all
                    words for the first object"
    :average-window 1)
```

`:average-window`

The values will be averaged over the last `:average-window` values. Default: `100`.

recording values

Values are recorded from event handlers. An example:

```
(define-event-handler
    (record-lexicon-of-first-agent-for-first-object
     interaction-finished)
  (loop with rules
     = ;; compute the rules of the first agent that have first
       ;; object of the world as meaning
     for rule in rules
     do (set-value-for-symbol monitor
                              (intern (word rule))
                              (rule-score rule))))
```

The function `(set-value-for-symbol monitor symbol value)` is used to store the rule score for each rule in the monitor, with symbol being the interned rule name. Each time you pass a value for a new symbol, a new list of recorded values is created. If you don't pass a value for some symbol during an interaction, the value `0` is recorded. Alternatively, you can use the function `(incf-value-for-symbol monitor symbol value)` to increase the current value for a symbol, starting in each interaction from `0`.

3.2.5.2 *monitor class* **alist-handler** *monitor*

description

The base class for all monitors that do something with data recorded by a `alist-recorder`.

`:recorder`	The id of the `alist-recorder` to use. That recorder will become automatically activated whenever a monitor derived form `alist-handler` gets activated.

3.2.5.3 *monitor class* **alist-printer** *alist-handler*

description — Prints the values of an alist recorder after each interaction.

example

```
(define-monitor print-lexicon-of-first-agent
    :class 'alist-printer
    :documentation "Prints for the first agent the scores of all
                    words for the first object"
    :recorder 'record-lexicon-of-first-agent-for-first-object
    :interval 100)
```

When this monitor is active, the output of a batch looks like this:

```
MY-EXPERIMENT> (run-batch *experiment* 1000 1)

100: vewiba: 0.50; sowape: 0.50;
200: vewiba: 0.50; sowape: 0.50;
300: vewiba: 0.50; sowape: 0.50;
400: vewiba: 0.60; sowape: 0.30;
500: vewiba: 0.80; sowape: 0.00; bofoxa: 0.10;
600: vewiba: 1.00; sowape: 0.00; bofoxa: 0.00; fapofo: 0.10;
700: vewiba: 0.90; sowape: 0.00; bofoxa: 0.00; fapofo: 0.00;
800: vewiba: 0.80; sowape: 0.00; bofoxa: 0.00; fapofo: 0.00;
900: vewiba: 0.90; sowape: 0.00; bofoxa: 0.00; fapofo: 0.00;
1000: vewiba: 0.80; sowape: 0.00; bofoxa: 0.00; fapofo: 0.10;
NIL
```

`:recorder`	See section 3.2.5.2.
`:interval`	The data is printed only every `:interval` interactions. Default: `1`.

3.2.5.4 *monitor class* **alist-gnuplotter** *alist-handler*

description — The base class for plotting data recorded by an `alist-recorder` with gnuplot.

Figure 3.4: An example for a real-time plot with a `alist-gnuplot-display` monitor

example

```
(define-monitor plot-all-words-for-first-object
    :class 'CLASS-DERIVED-FROM-ALIST-GNUPLOTTER
    :documentation "Plots scores of all words for the first object"
    :recorder 'record-all-words-for-first-object
    :minimum-number-of-data-points 500
    :error-bars nil
    :key-location "below"
    :y-min 0 :y-max 1.05 :draw-y-grid t
    :y-label nil x-label nil
    :line-width 1 :draw-y-grid t
    :colors '("red" "blue" "black" "green" "gold"))
```

Depending on what `CLASS-DERIVED-FROM-ALIST-GNUPLOTTER` is, the result might look like in figure 3.4 or figure 3.5.

`:recorder`	See section 3.2.5.2.

`:minimum-number-of-data-points`
`:errorbars`
`:colors`
`:line-width`
`:key-location`
	See section 3.2.4.1

`:y-min :y-max`	The minimum and maximum y values. When not provided, the graph is automatically scaled (which is often better).

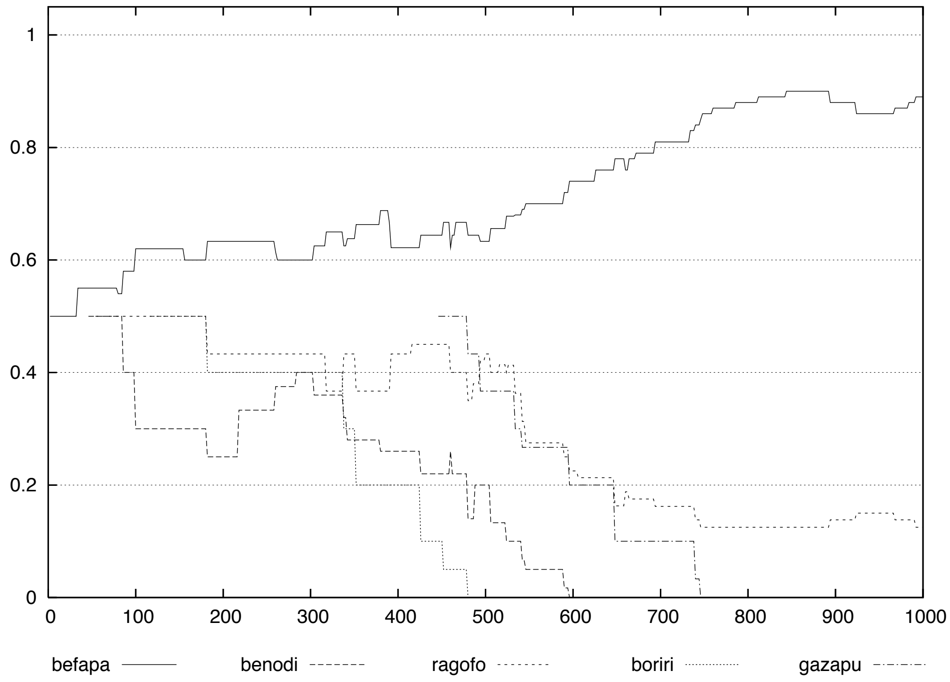

Figure 3.5: An example for a plot generated by a `alist-gnuplot-graphic-generator` monitor

`:draw-y-grid`	When `t`, thin horizontal lines are drawn at the height of the y ticks.

3.2.5.5 *monitor class* **alist-gnuplot-display** *alist-gnuplotter*

description: Displays plots of data recorded by an `alist-recorder` in real-time.

example:

```
(define-monitor display-all-words-for-first-object
    :class 'alist-gnuplot-display
    :documentation "Plots scores of all words for the first object"
    :recorder 'record-all-words-for-first-object
    :draw-y-grid t :y-max 1.05 :y-min 0
    :update-interval 25)
```

The resulting graph looks as in figure 3.4.

`:update-interval`	How often the display is redrawn. Default: `100`.

All other parameters are as described in section 3.2.5.4.

3.2.5.6 *monitor class* **alist-gnuplot-graphic-generator** *alist-gnuplotter*

description	Produces a graph file at the end of a batch. It is recommended to use such a graph for papers, as they are black and white.

example

```
(define-monitor plot-all-words-for-first-object
    :class 'alist-gnuplot-graphic-generator
    :documentation "Plots scores of all words for the first object"
    :recorder 'record-all-words-for-first-object
    :draw-y-grid t :y-max 1.05 :y-min 0
    :file-name (babel-pathname :directory (list "tmp")
                               :name "lexicon" :type "ps")
    :add-time-and-experiment-to-file-name nil
    :graphic-type "postscript")
```

The resulting file */tmp/lexicon.ps* looks as in figure 3.5.

`graphic-type`	Which gnuplot graphic driver to use. Should be one out of `"postscript"`, `"pdf"`, `"svg"` or `"gif"`.
`:file-name`	The file name of the graphic file to produce. Should be a `pathname`.
`:add-time-and-experiment-to-file-name`	See section 3.2.3.4. All other parameters are the same as in section 3.2.5.4.

3.2.5.7 *monitor class* **alist-gnuplot-display-and-graphic-generator** *alist-gnuplot-display alist-gnuplot-graphic-generator*

description	A monitor that produces a real-time plot and generates a graphic file in the end. As it derives from both of these classes, it takes all parameters that `alist-gnuplot-display` and `alist-gnuplot-graphic-generator` take.

3.3 Behind the Scenes

As mentioned in the beginning of section 3.2, you will normally not get in touch with the classes that form the base of the monitor system. Instead, you will use the macros from section 3.1. However, if you want to define some monitors that do something else than what you can do with the built-in monitors and if you want to reuse this functionality, you might want to derive your own monitor classes.

This section describes the monitor, event, and event handling base classes and methods that are defined in *systems/monitors/base.lisp*. You can easily skip this section if you are only interested in using the monitors.

Please note that all the classes and methods are not exported from the `:monitors` package as they are “hidden” from the user. It is recommended to put your own classes in that package too.

3.3.1 Classes for Monitors and Events

3.3.1.1 *class* **monitor**

description	The base class for all monitor classes. Stores the events that a monitor is subscribed to and whether the monitor is active.
slot `id`	A unique id.
slot `event-ids`	The ids of events that the monitor is “subscribed” to. Technically, these are all events for that an event handler was defined.
slot `active`	If `t`, then this monitor is notified on its events.
slot `documentation`	A string that helps the user to guess the purpose of the monitor.
slot `source-file`	The file in that the monitor was defined. This is guessed using the LISP variable `*load-pathname*`, which not always contains a file name.
slot `init-arguments`	The keyword parameter list with that the monitor was defined. This is stored in order to be able to redefine a monitor only when its parameters change.
slot `error-occured-during-initialization`	When `t`, an error occured during the initialization. This information is stored for determining when to redefine a monitor.

3.3.1.2 *class* **event**

description	Represents an event with its parameters and a list of monitors that are listening to the event.
slot `id`	An unique event id.
slot `active-monitors`	A list of the ids of those monitors that subscribed to this event and that are active.
slot `source-file`	The file in that the event was defined. This is guessed using the LISP variable `*load-pathname*`, which not always contains a file name.
slot `parameters`	A list of (name type) parameter definitions, used for method generation.

3.3.1.3 *global variable* ***monitors***

description	A hash table containing the instances of all defined monitors. When a monitor is defined with `define-monitor`, the instance is automatically stored there. Due to that, the user does not have to care about keeping pointers to these instances and it is easier to access a particular monitor, for example for activating it.

3.3.1.4 *global variable* ***events***

description	A hash table containing the instances of all defined events.

3.3.1.5 *function* **get-monitor** *id*

description	Returns the monitor instance for an `id`. Normally you will never do this but it helps for example when you want to see what kind of data a `data-recorder` recorded.
`id`	The id of the monitor.
example	`(monitors::get-monitor 'record-number-of-lex-stem-rules)`

3.3.1.6 *function* **get-event** *id*

description	Returns the event instance for an `id`.
`id`	The id of the event.

3.3.2 The Creation of Monitors, Events and Handlers

To prevent the user from being confronted with the classes above and in order to guarantee some properties of monitors and events, there is a lot of checking going on. In `:around` methods of `initialize-instance`, it is checked for almost every parameter whether the passed values are correct. You will normally get helpful error messages if you passed something wrong.

For convenience, when a source file is recompiled or reloaded, the state of the contained monitors and events is preserved. For that, monitors copy the state of previous instances in the `initialize-instance :around` methods. Additionally, monitors and events are only redefined when there is no previous instance or when parameters changed:

3.3.2.1 *function* **make-monitor-unless-already-defined** *id class &optional init-arguments*

description	Makes a new monitor when there is (1) no previously defined monitor with the same `id`, (2) the arguments passed to the monitor (`init-arguments`) are different from a previous monitor with the same id, or (3) an error occurred during the initialization of the previous monitor.
`id`	The id of the monitor.
`class`	The class to instantiate.
`init-arguments`	All keyword parameters that are needed to make the instance.

3.3.2.2 *function* **subscribe-to-event** *monitor-id event-id*

description	Adds an event to the `event-ids` list of a monitor. When the monitor is a active, it is also added to the `active-monitors` slot of the event.
`monitor-id`	The id of the monitor.
`event-id`	The id of the event.

3.3.2.3 *function* **make-event-unless-already-defined** *id parameters*

description Makes an event (only when it is not defined yet or when parameters changed). Additionally, it defines a generic function for handling the event. For example for event `interaction-started`, this generic function is generated, incorporating the passed parameters:

```
(defgeneric handle-interaction-started-event
    (monitor monitor-id event experiment))
```

There is a default implementation that specializes on `t` for the parameters of the event. If you type for example `(notify interaction-started 42)`, then that method will give you this error:

```
Parameter 2 (EXPERIMENT) should be of type EXPERIMENT.
Instead, 42 (FIXNUM) was passed.
   [Condition of type SIMPLE-ERROR]
```

`id`	The id of the event.
`parameters`	The parameter list.

3.3.2.4 *function* **make-event-handler** *monitor-id event-id body*

description	Subscribes a monitor to an event (using `subscribe-to-event`, section 3.3.2.2) and creates an event handler method.
`monitor-id`	The id of the monitor. Either a symbol or a list of symbols (for defining multiple handlers at the same time).
`event-id`	The id of the event to handle.
`body`	The body of the handler method (see section 3.1.3).

In order to put events and their handlers into the same source file, the events and monitors have to be defined at macro expansion time (the macro expansion of the `notify` macro (section 3.1.4 needs to know the parameters of the event to generate the method call). But as there is no macro expansion when a compiled file is loaded, the events also have to be defined at load time.

This problem is solved by defining each event and monitor twice:

```
(defmacro define-event (id &rest parameters)
  (make-event-unless-already-defined id parameters)
  `(make-event-unless-already-defined ',id ',parameters))
```

The first line is executed during macro expansion when a source file is compiled. The second line will be executed as well but does not do anything because the event is already defined. When loading a compiled file, only the second line was compiled into the code and is executed. The same thing happens in the `define-monitor` macro.

3.3.3 Defining own Monitor Classes

This section shows how to derive own monitor classes. The existing class `trace-monitor` (section 3.2.1.1) serves as an example. The first thing to do is to derive an own class from one of the built-in monitor classes and define some slots that the monitor will need to function (there are no slots needed in this example):

```
(defclass trace-monitor (monitor)
  ()
  (:documentation "Prints string messages on a screen or keeps them in a shared buffer
                   for later retrieval"))
```

The next thing is to define the initialization of the monitor instance. This typically looks like this:

```
(defmethod initialize-instance :around ((monitor trace-monitor)
                                        &key id &allow-other-keys)
  (let ((previous-monitor (get-monitor id)))
    (call-next-method)
    (setf (error-occured-during-initialization monitor) t)
    (make-event-unless-already-defined id '((message string)))
    (when (or (not previous-monitor)
              (error-occured-during-initialization previous-monitor)
              (not (find id (event-ids monitor))))
      (make-event-handler
       id id `((unless (equal ,(intern-in-package-of id "MESSAGE") "")
                 (format (monitor-stream monitor) "~a"
                         ,(intern-in-package-of id "MESSAGE")))))))
  (setf (error-occured-during-initialization monitor) nil))
```

Before the base class is initialized with `(call-next-method)`, a reference to the old monitor instance is kept in variable `previous-monitor`. Then, the monitor slot `error-occured-during-initialization` is set to `t`. This is set to false again in the last line. If something goes wrong between these two lines, the value of that slot remains `t` so that the next attempt to initialize starts from scratch again.

Then there is automatically an event defined that has the same name as the monitor and takes a string as parameter. Only when there is no previous monitor, when there were no errors and when there is no handler yet for that automatically generated event, an event handler that prints or buffers the message is defined using the `make-event-handler` macro.

4 The Experiment Framework

This chapter defines what an "experiment" is in the Babel framework. There is a "population" of "agents" that engage in "interactions". An agent is understood as a software entity that interacts with a "world" (see for example Russel and Norvig, 1995) has an internal state, own goals and means to achieve them. It perceives information from the world and performs "actions" on it. It has ways to *diagnose* problems in its information processing and *repair strategies* to adapt and optimize its state and processing. Agents can not look into each others brains, which means that they never can access the internal states of others. Instead, they perform actions such as speaking or pointing that are observed by the other agents. An experiment itself is a controlled repetition (*batch*) of *series* of such interactions. For each series, measures of the emergent behavior or properties of the agent's information processing are recorded.

The experiment framework defines[1] the concepts introduced above in a rather abstract way. Specific kinds of experiments such as different types of language games are operationalized by subclassing and implementing generic methods described in this chapter.

4.1 Agents Situated in the World

This section defines agents and their interaction with the world.

4.1.1 Actions Performed on the World

Actions are performed by agents. They can change the state of the world and are observable by other agents. They can be anything that a robot or a human could do, for example speaking, pointing, giving, walking, nodding, etc – but rather no telepathic or other supernatural activities.

4.1.1.1 *class* **action**

description	Represents an action performed by an agent. You define own actions by subclassing from `action`.
slot `agent-id`	`(agent-id :type (or symbol fixnum) :initform nil :initarg :agent-id :accessor agent-id)` The id of the agent that does the action. Normally you will not have to provide this when creating an action because it is set automatically in an `:around` method of `run-agent` (see section 4.1.2.2).

---

[1] See also the source files in directory *systems/experiment-framework*.

slot `recipient-ids`

```
(recipient-ids :type list :initform '(all-agents)
               :initarg :recipient-ids :accessor recipient-ids)
```

The ids of the agents that the action is directed to. By default, the action is performed to all interacting agents (value `'(all-agents)`).

4.1.1.2 *class* **no-action** *action*

description

When an agent performs this action it means that it waits for other agents to do something or that it believes the current interaction to be finished.

4.1.1.3 *class* **world**

description

The state of the shared world in which the agents are interacting. It is highly experiment dependent what that is. Normally you will define a world for your experiment by subclassing from `world`.

slot `actions`

```
(actions :type list :initform nil :accessor actions)
```

All actions that were performed by the agents during the current interaction. Newer actions are first in the list. These are added automatically by the framework after update-world (see below).

4.1.1.4 *generic function* **initialize-world-for-next-interaction** *world*

description

Initializes or updates the state of the world at the beginning of a new interaction. It is called automatically in an `:around` method of `run-interaction` (see section 4.2.2.3).

`world`

The world to initialize.

default implementation

The default implementation is empty, because there might be experiments where the world does not have state except the actions performed by the agents. In a `:before` method, the `actions` slot of the world is set to `nil`.

4.1.1.5 *generic function* **update-world** *world action*

description

Updates the world dependent on the last action of an agent. It is automatically called from `run-interaction` (see section 4.2.2.3). If in your experiment the agents perform actions that can change the state of the world, then you would implement a method for your `world` and `action` class.

`world`

The world to update.

`action`

The action

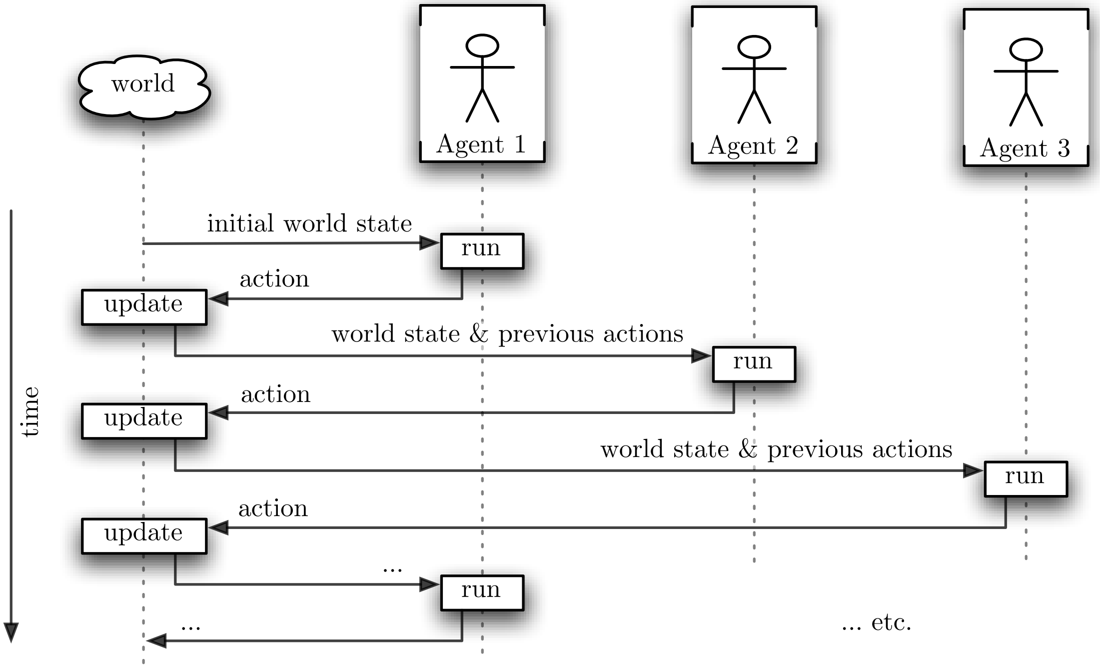

Figure 4.1: The general scheme of an interaction between agents. All agents that participate in the interaction are run one after the other, observe the state of the world and the actions that were performed by the other agents. As a result, they return an action on which again updates the world. An interaction ends when all agents return the no-action.

default implementation	The default implementation is empty. This means that actions such as speaking or nodding do not change the state of the world but are directly observed by the agents. In an `:around` method, the `action` is automatically pushed on the `actions` slot of the `world`.

4.1.2 Running Agents

The agents that take part in an interaction are run one after the other to determine their next action (see figure 4.1).

4.1.2.1 *class* **agent** *object-with-learning-mechanisms*

description	The base class for all agents. It inherit slots for learning mechanisms from base class `object-with-learning-mechanisms` (see section 4.3).

slot `id`

```
(id :initarg :id :reader id :type (or symbol fixnum)
    :initform (gen-sym))
```

The id of the agent.

slot `problems`

```
(problems :type list :initform nil :initarg :problems
          :accessor problems)
```

All the problems that were encountered during an interaction.

slot `list-of-interacting-agents`

```
(ids-of-interacting-agents :initform nil :type list
                           :accessor ids-of-interacting-agents)
```

The ids of all other agents that are part of the current interaction (see section 4.2.2.1).

4.1.2.2 *generic function* **run-agent** *agent world*

description: Plans and performs the next action of an agent. Called by `run-interaction` (see section 4.2.2.3).

`agent`: The agent to perform the action.

`world`: The state of the world including the actions performed by the other agents.

default implementation: The default implementation calls repeatedly

- `plan-action` (see section 4.1.2.3 below)
- `run-agent-diagnostics` (see chapter 5)
- and `run-agent-repair-strategies` (see chapter 5)

in a `loop` until nothing is repaired anymore by the repair strategies (`run-agent-repair-strategies` returns `nil`). Then the last planned action is passed to `perform-action` (see section 4.1.2.5) and the action returned by that is returned by `run-agent`.

For example there could be an agent that tries to interpret an utterance. First `plan-action` runs an interpretation task that fails and returns two values, the returned planned action which is a ‘signal failure’ action and a list of agent level learning situations, e.g. (agent-interpreting). Based on these learning situations there are learning mechanisms that repair the agent. Again `plan-action` is run and this time returns a ‘signal success’ action. This action is passed to `perform-action`, which commits the things previously learned and updates its linguistic inventory based on the communicative success.
In an `:around` method, there are notifications for the events `run-agent-started` and `run-agent-finished` (sections 4.1.2.6 and 4.1.2.7).

4.1.2.3 *generic function* **plan-action** *agent world*

description	Plans an action for the agent based on the world (which contains the actions performed by the other agents). Returns an instance of `action` and a list of agent level learning situations for that the agent level learning mechanisms are run. This function is called from `run-agent`, which runs learning mechanisms after it and, if something is repaired, runs `plan-action` again. So the action you return here is not necessarily the one that is returned by `run-agent`. If you want to do something based on the action that the agent actually performs, then you can do that in method `perform-action` (see below).
`agent`	The agent that is run.
`world`	The world that the agent is situated in.
default implementation	There is a default implementation that calls `plan-action-based-on-last-action` (see below) on the last action of the world. However, this might be not a good idea when there are more than two agents interacting and when actions depend on more than just the action of the last agent that was run. In this case you might consider re-implementing this method for your `agent` class.

4.1.2.4 *generic-function* **plan-action-based-on-last-action** *agent world last-action*

description	Returns an action based on a single last action performed by an (the) other agent. When you use the default implementation of `plan-action`, then you have to implement this method for all actions that your agents can perform. Just as plan-action it also has to return a list of learning situations as second value.
`agent`	The agent that performs the action.
`world`	The world in that the agent is situated.
`last-action`	The last performed action of a (the) other agent.

4.1.2.5 *generic function* **perform-action** *agent planned-action*

description	Called by the default implementation of `run-agent` on the planned action after nothing is repaired anymore. Returns an action. This gives you the chance to do something based on the action that the agent performs.
`agent`	The agent that performs the action.
`planned-action`	The action that was returned by the last call to `plan-action`.
default-implementation	The default implementation simply returns the `planned-action`.

4.1.2.6 *monitor event* **run-agent-started** *(agent agent) (world world)*

description	Triggered at the begin of `run-agent`.
`agent`	The agent that is run.
`world`	The world in the agent is situated.

4.1.2.7 *monitor event* **run-agent-finished** *(agent agent) (world world) (action action)*

description	Notified at the end of `run-agent`.
`agent`	The agent that performed an action.
`world`	The world (not yet updated on the action).
`action`	The performed action.

4.1.2.8 *generic function* **initialize-interaction** *agent*

description/ default implementation	Is called for each of the interacting agents at the begin of an interaction. For example can be used to initialize the role that an agent takes in the interaction. The default implementation is empty.
`agent`	The agent to initialize.

4.1.2.9 *generic function* **consolidate-agent** *agent*

description/ default implementation	Is called for each of the interacting agents at the end of an interaction. It is intended to be used for committing learned things or updating scores of inventories. The default implementation is empty.
`agent`	The agent to consolidate.

4.2 Interacting Agents

The main purpose of the experiment framework is to have agents interacting with each other and learn from that. This section describes how these interactions are defined and run.

4.2.1 Experiments and Populations

Experiments determine how interactions between agents of a population are run.

4.2.1.1 *class* **experiment** *object-with-learning-mechanisms*

description

The base class for all experiments. It also contains learning mechanisms as it is derived from `object-with-learning-mechanisms` (see section 4.3).

slot `population`

```
(population :type list :accessor population)
```

A list of agents. The population is automatically initialized when the experiment is created (using the generic function `initialize-population`, see below).

slot `interaction-number`

```
(interaction-number :type fixnum :initform 0
                    :accessor interaction-number)
```

A counter that is increased with every interaction.

slot `interacting-agents`

```
(interacting-agents :type list :initform nil
                    :accessor interacting-agents)
```

A list of the agents that are involved in the current interaction. Determined by function `determine-interacting-agents` (see section 4.2.2.1).

slot `processing-strategies`

```
(processing-strategies :type t :initform t
                       :accessor processing-strategies
                       :initarg :processing-strategies)
```

An experiment specific object that can be used to specialize methods on lower levels. These can be methods that you have to implement for your own experiment. There are also built-in methods that provide multiple options for some particular processing. By deriving your processing strategy from

slot `world`

```
(world :type world :initarg :world :accessor world
       :initform (make-instance 'world))
```

The world that is shared by the agents.

4.2.1.2 *generic function* **initialize-population** *experiment*

description

Replaces all agents of an experiment's population by a new list of agents. The new list should contain at least one agent. In an `:after` method, all learning mechanisms and processing mechanisms of the experiment are automatically copied into each agent. Additionally, there is a notification for event `population-initialized` (see below).

You have to implement this method for your experiment.

example

```
(defclass my-experiment (experiment) ())
(defclass my-agent (agent) ())

(defmethod initialize-population ((experiment my-experiment))
  (setf (population experiment)
        (loop for n from 1 to 5
          collect (make-instance 'my-agent :id n)))
```

4.2.1.3 *monitor event* **population-initialized** *experiment*

description	Triggered after `initialize-population`.
`experiment`	The experiment for that the population was initialized.

4.2.2 Running an Interaction

An "interaction" is when some agents are drawn from the population and interact for some time in a shared world. This could be for example a guessing game where one agent of the population becomes a speaker, describes a scene to a hearer, the hearer points to a thing in the world, and the speaker signals the communicative success of the interaction.

4.2.2.1 *generic function* **determine-interacting-agents** *experiment*

description	Called at the begin of each interaction to determine which agents (a subset of the population) will interact with each other. The function has to set the `interacting-agents` slot of `experiment`.
`experiment`	An experiment instance.
default implementation	The default implementation randomly selects two agents from the population of `experiment`. In an `:around` method, the `ids-of-interacting-agents` slots of all interacting agents are set properly so that each agent knows with who it is interacting. There is a notification for event `interacting-agents-determined` (see below).

4.2.2.2 *monitor event* **interacting-agents-determined** *(experiment experiment)*

description	Triggered at the end of `determine-interacting-agents`.
`experiment`	The experiment.

4.2.2.3 *generic function* **run-interaction** *experiment*

description	The most important function of the experiment framework. Runs one interaction of an experiment.
`experiment`	The experiment to run.

default implementation

It is not recommended to specialize this function for your experiment class as there is already a sophisticated default implementation for class `experiment` itself. The default implementation calls the `run-agent` method (see section 4.1.2.2) of the first interacting agent. The returned action is passed to `update-world` (section 4.1.1.5). Then the `run-agent` method of the next agent is called and so on until no agent returns an action different from `no-action` (section 4.1.1.2). See also figure 4.1.

```
(defmethod run-interaction ((experiment experiment))
  (loop for at-least-one-agent-returned-an-action = nil
     do (loop for agent in (interacting-agents experiment)
              for action = (run-agent agent (world experiment))
              do (unless (typep action 'no-action)
                   (setf at-least-one-agent-returned-an-action t))
                (update-world (world experiment) action))
     while at-least-one-agent-returned-an-action))
```

In an `:around` method,

- the `interaction-number` of the experiment is increased,
- there is a notification on the event `interaction-started` (see section 3.1.11.1),
- the method `called-before-run-interaction` is called (see below),
- `determine-interacting-agents experiment` is called (see above),
- for each interacting agent `initialize-interaction` (section 4.1.2.8) is called,
- the world is initialized with `initialize-world` (section 4.1.1.4),
- the main method is called,
- for each interacting agent `consolidate-agent` (section 4.1.2.8) is called,
- `called-after-run-interaction` (see below) is called,
- there is a notification for event `interaction-finished` (see 3.1.11.2).

4.2.2.4 *generic function* **called-before-run-interaction** *experiment*

description

The method is called automatically before `run-interaction`.

`experiment`

The experiment instance.

default implementation

There is a default implementation that does nothing. You can, but don't have to, do additional things here such as initializing variables, interacting with robots, etc.

Of course you could also write a `:before` method for `run-interaction` (it would be the same), but maybe implementing this method for your experiment class makes things look more clear.

4.2.2.5 *generic function* **called-after-run-interaction** *experiment*

description	The same as `called-before-interaction`. except that it is called automatically after `run-interaction`.
`experiment`	The experiment instance.

4.2.3 Running Experiments

There are three different levels of running an experiment:

- An interaction, see previous section 4.2.2.
- A "series" is a set of subsequent interactions, possibly with different agents of the population participating in the interaction each time.
- A "batch" is a repetition of series with same length. Before each series, the population is reset. This is enables you to average experimental results over many repetitions.

4.2.3.1 *generic function* **run-series** *experiment number-of-interactions &key reset*

description	Runs a series of interactions.
`experiment`	The experiment instance.
`number-of-interactions`	How many interactions to run.
`:reset`	Whether to reset the population and the monitors. Default: `nil`.
default implementation	The default implementation executes the `run-interaction` method `number-of-interaction` times. When `:reset` is `t`, in the beginning it sets the `interaction-number` of the experiment to `0`, resets the monitors by notifying on the event `reset-monitors` (section 3.1.11.5) and calls the `initialize-population` method (section 4.2.1.2).

4.2.3.2 *generic function* **run-batch** *experiment number-of-interactions number-of-series*

description	Runs a batch (multiple series of interactions).
`experiment`	The experiment instance.
`number-of-interactions`	How many interactions to run.
`number-of-series`	How many series to run.

default implementation	The default implementation resets the monitors in the beginning. Then it runs `number-of-series` times a series of `number-of-interactions` interactions, each time in the beginning setting the `interaction-number` to 0 and resetting the population. After each series, the event `series-finished` is triggered. In the end, there is a notification for `batch-finished`.

4.2.3.3 *monitor* **trace-interaction** *trace-monitor*

description	Prints information about interactions such as the interaction number, which agent is run and which actions it returned.

4.2.3.4 *monitor* **trace-experiment** *trace-monitor*

description	Prints information about the experiment, when the population is reset and when a series or a batch is finished. Good for observing the progress in large-scale simulations.

4.3 Learning Mechanisms

Learning mechanisms are deeply grounded into the Babel framework. They consist of three things:

1. “diagnostics” reside on top of cognitive processes and try to detect failures or suboptimal processing, which they report in the form of
2. “problems”. They contain the information that can be used by
3. “repair strategies” to overcome failures or to optimize information processing.

Since learning is crucially important we have dedicated a chapter to accommodate all there is to know about learning in the Babel framework. We refer you to chapter 5

4.4 An Interaction Example

This section shows a very minimal experiment without learning mechanisms that illustrates the concepts described in this chapter. An interaction in this experiment will be that one agent asks for chocolate, gets chocolate and thanks the other agent for it.

First, we define an experiment class, an agent class and an initialization of the population:

```
(defclass my-experiment (experiment) ())
(defclass my-agent (agent) ())

(defmethod initialize-population ((experiment my-experiment))
  (setf (population experiment)
        (loop for i from 1 to 5 collect (make-instance ’my-agent :id i))))
```

We use the default implementation of `determine-interacting-agents` (which randomly selects two agents of the population). We define the two actions that we need for this interaction:

```
(defclass give-action (action)
  ((object :initarg :object :accessor object)))

(defclass speak-action (action)
  ((utterance :initarg :utterance :accessor utterance)))
```

Then we implement the `plan-action-based-on-last-action` methods for `my-agent` and the actions above:

```
(defmethod plan-action-based-on-last-action ((agent my-agent) (world world)
                                             (last-action (eql nil)))
  (make-instance 'speak-action :utterance "Want chocolate!"))

(defmethod plan-action-based-on-last-action ((agent my-agent) (world world)
                                             (last-action speak-action))
  (if (string-equal (utterance last-action) "Want chocolate!")
      (make-instance 'give-action :object 'chocolate
                     :recipient-ids (list (first (ids-of-interacting-agents agent))))
      (make-instance 'no-action)))

(defmethod plan-action-based-on-last-action ((agent my-agent) (world world)
                                             (last-action give-action))
  (make-instance 'speak-action :utterance "Thank you!"))

(defmethod plan-action-based-on-last-action ((agent my-agent) (world world)
                                             (last-action no-action))
  (make-instance 'no-action))
```

The first `plan-action-based-on-last-action` method above is run at the begin of an interaction (there are no last actions). The agent that is run first says “want chocolate”. Then the next agent is run. If it hears “want chocolate!”, then it gives chocolate to the other agent. An agent that gets chocolate says “Thank you!”. Whenever the other agent did nothing, an agent also does nothing.

This makes an instance of the experiment:

```
(defparameter *experiment* (make-instance 'my-experiment))
```

To see some output, we activate the monitor `trace-interaction` and then run one interaction:

```
(activate-monitor trace-interaction)

(run-interaction *experiment*)
```

This is the output:

```
=======================
= Started interaction 1.
= Interacting agents: (<my-agent 3> <my-agent 4>)
======================
```

```
= Running <my-agent 3>.
= <my-agent 3> performs
   <speak-action: utterance: "Want chocolate!"
    <action: agent-id: 3, recipient-ids: (ALL-AGENTS)>>.
======================
= Running <my-agent 4>.
= <my-agent 4> performs
   <give-action: object: CHOCOLATE <action: agent-id: 4, recipient-ids: (3)>>.
======================
= Running <my-agent 3>.
= <my-agent 3> performs
   <speak-action: utterance: "Thank you!"
    <action: agent-id: 3, recipient-ids: (ALL-AGENTS)>>.
======================
= Running <my-agent 4>.
= <my-agent 4> performs
   <no-action: <action: agent-id: 4, recipient-ids: (ALL-AGENTS)>>.
======================
= Running <my-agent 3>.
= <my-agent 3> performs
   <no-action: <action: agent-id: 3, recipient-ids: (ALL-AGENTS)>>.
======================
= Running <my-agent 4>.
= <my-agent 4> performs
   <no-action: <action: agent-id: 4, recipient-ids: (ALL-AGENTS)>>.
```

4.5 Running Parallel Series of Experiments

Experiments that involve grammar learning, rich conceptualization mechanisms or excessive search can be very slow or can require large number of interactions for the desired pheomeon to emerge. Especially running multiple series of the same experiment in order to have averaged results (running batches) can take forever on a single processor.

Our experiment framework provides one mechanism for speeding this up: multiple series of the same experiment can be run in parallel on a machine that has multiple processors. So far this works only on SBCL and only on machines where different sub-processes are automatically scheduled, but it is very likely that other Lisps and machine architectures will be supported in the future.

Furthermore, there are functions for analyzing the impact of different configurations on a particular measure within one graph.

The functions for parallel batches are defined in *systems/experiment-framework/parallel-batch.lisp*.

4.5.1 *function* **run-parallel-batch** *&key ...*

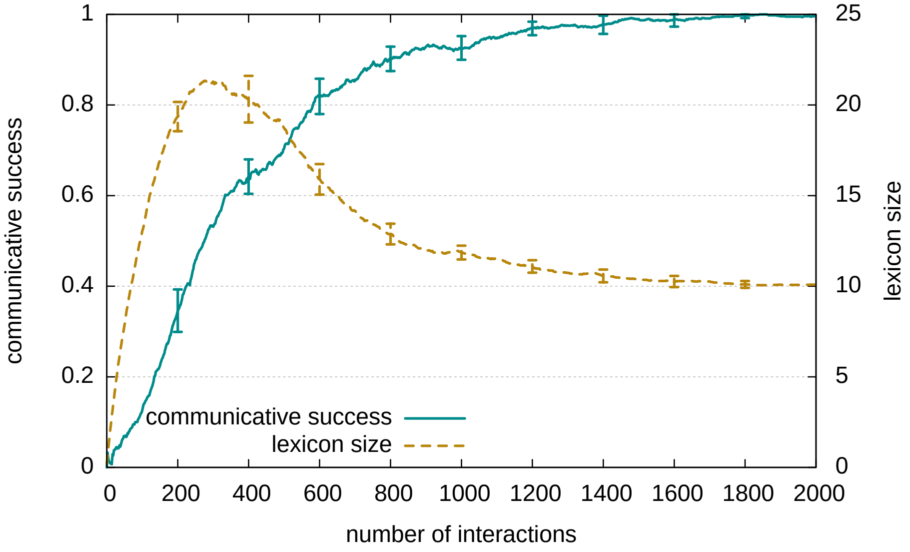

Figure 4.2: An example for a graph created with `run-parallel-batch`. This is the same graph that also would be created with `run-batch`, except that it is faster to use `run-parallel-batch`.

description	The same as method `run-batch` (see section 4.2.3.2), with the difference that each series is run in parallel. Internally it starts for each of the series a separate client Lisp process and then loads and runs a specified experiment in it. The results of each series are then collected and graphs are produced.
`:asdf-system`	The asdf system of the experiment to run.
`:package`	The package to run the experiment in.
`:experiment-class`	The experiment to run.
`:number-of-interactions`	How many interactions to run in each client.
`:number-of-series`	How many series to run. This also determines also how many client Lisp processes will be started. When for example your machine has 10 processors and `:number-of-series` is 10, then each of these processes will be run at 100% CPU usage, speeding up the batch to take only 10% of the time of a serial batch on a single processor. When there are only 5 processors, then each Lisp will run at 50%, speeding up the batch to take 20% of the time.
`:monitors`	Which monitors to use (a list of strings). This can be any kind of monitor, but it makes sense only for those that use data recorders, e.g. gnuplot monitors or data writers.

example

```
(run-parallel-batch
   :asdf-system "babel-demo"
   :package "babel-demo"
   :experiment-class "naming-game"
   :number-of-interactions 2000
   :number-of-series 10
   :monitors '("babel-demo::plot-success+lexicon-size"))
```

This has the same effect as running

```
(asdf:operate 'asdf:load-op :babel-demo)

(in-package :babel-demo)

(activate-monitor babel-demo::plot-success+lexicon-size)

(run-batch (make-instance 'naming-game) 2000 10)
```

, except that it takes only 10% of the time. The resulting graph is shown in figure 4.2.

4.5.2 *function* **create-graphs-for-different-experimental-conditions** *&key ...*

description Creates graphs for the impact on different experimental conditions on particular measures. Different conditions are implemented as separate experiment classes that are then run one after each other, with the results merged into one graph.

example When there is for example an experiment class `naming-game`, different conditions that analyze different strategies for lexicon update can be created by subclassing from `naming-game` and then setting different configurations during the initialization:

```
(defclass ng-1 (naming-game) ())
(defclass ng-2 (naming-game) ())
(defclass ng-3 (naming-game) ())

(defmethod initialize-instance :after ((experiment ng-1) &key)
  (set-configuration experiment 'word-score-delta-success 0.0)
  (set-configuration experiment 'word-score-delta-inhibit 0.0)
  (set-configuration experiment 'word-score-delta-fail 0.0))

(defmethod initialize-instance :after ((experiment ng-2) &key)
  (set-configuration experiment 'word-score-delta-success 0.1)
  (set-configuration experiment 'word-score-delta-inhibit 0.0)
  (set-configuration experiment 'word-score-delta-fail -0.1))

(defmethod initialize-instance :after ((experiment ng-3) &key)
  (set-configuration experiment 'word-score-delta-success 0.1)
  (set-configuration experiment 'word-score-delta-inhibit -0.2)
  (set-configuration experiment 'word-score-delta-fail -0.1))
```

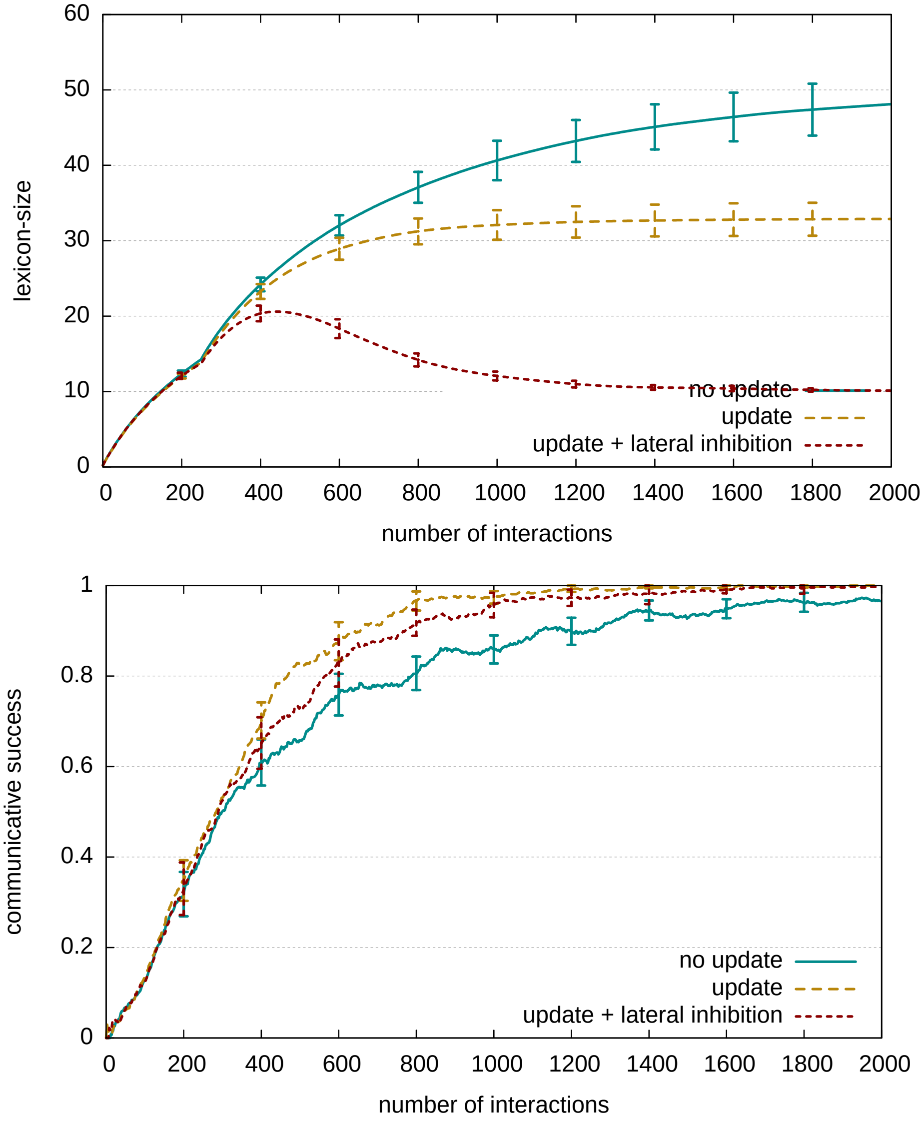

Figure 4.3: Examples for graphs create by `create-graphs-for-different-experimental-conditions`.

The impact of these different update strategies can then be plotted with this:

```
(create-graphs-for-different-experimental-conditions
   :asdf-system "babel-demo"
   :package "babel-demo"
   :experiment-base-class "naming-game"
   :experiment-classes '("ng-1" "ng-2" "ng-3" )
   :captions '("no update" "update" "update + lateral inhibition")
   :number-of-interactions 2000
   :number-of-series 10
   :data-recorders '("babel-demo::record-communicative-success"
     "babel-demo::record-average-number-of-words")
   :average-data '(t t)
   :parameters-for-graphic-generators
   '((:x-label "number of interactions" :y1-label "communicative success"
      :error-bars t :y1-max 1 :y1-min 0 :draw-y1-grid t
      :graphic-type "pdf" :key-location "right bottom"
      :file-name
        (babel-pathname
          :name "success-vs-update-strategy" :type "pdf"
          :directory '("experiments" "babel-demo" "graphs")))
     (:x-label "number of interactions" :y1-label "lexicon-size"
      :error-bars t :y1-min 0 :draw-y1-grid t
      :graphic-type "pdf" :key-location "right bottom"
      :file-name
        (babel-pathname
          :name "lexicon-size-vs-update-strategy" :type "pdf"
          :directory '("experiments" "babel-demo" "graphs")))))
```

The resulting-graphs are shown in figure 4.3.

`:asdf-system :package :number-of-interactions :number-of-series`	The same as in `run-parallel-batch` above.
`:experiment-base-class`	The base class of the different conditions (only needed for creating graph file names).
`:experiment-classes`	The experiments to run (a list of strings).
`:captions`	A graph caption for each of the conditions.
`:data-recorders`	The measures (data recorders) to use. For each measure a separate graph will be created.
`:average-data`	Determines for each of the measures whether the results are averaged or not.
`:parameters-for-graphic-generators`	Specifies the appearance of the graphs. Internally, instances of `gnuplot-graphic-generator`s are created and everything in `:parameters-for-graphic-generators` is passed to them, so for details see section 3.2.4.3.

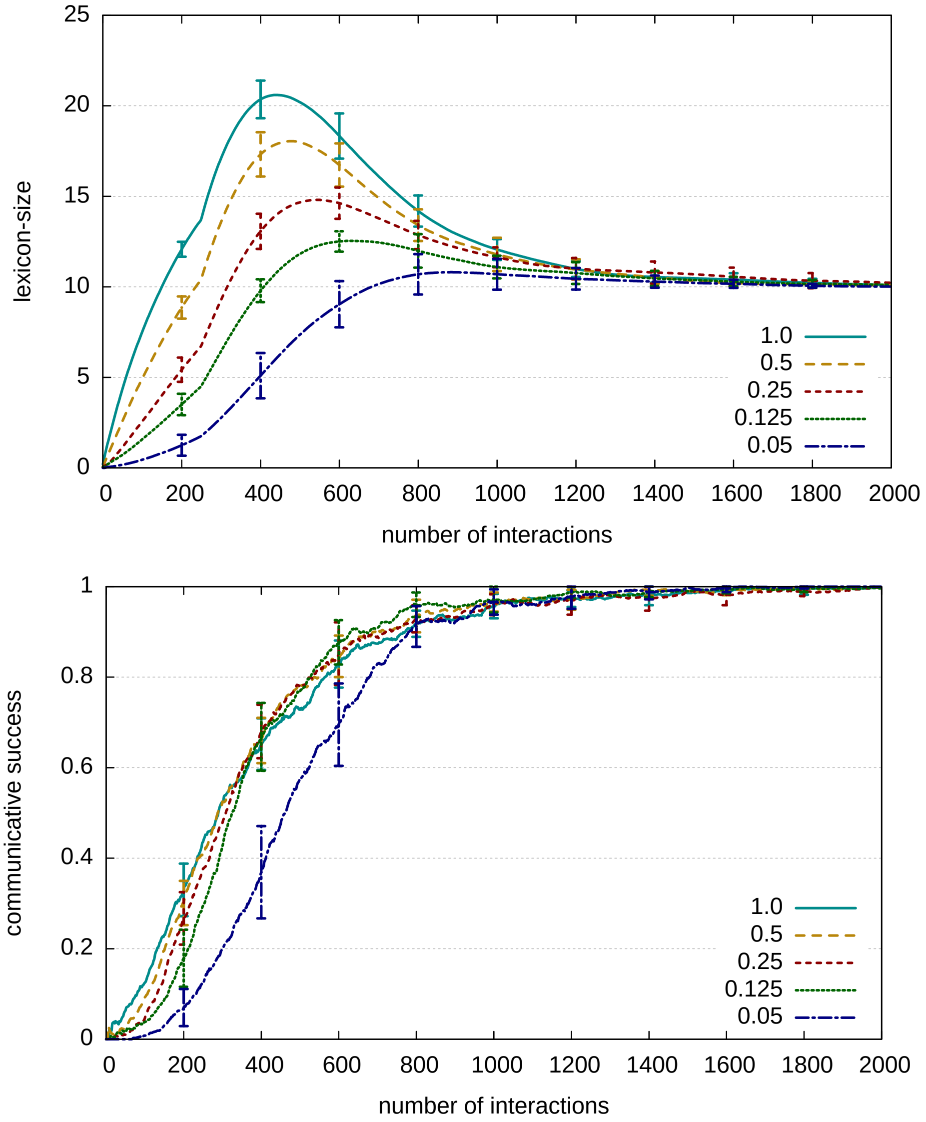

Figure 4.4: Examples for graphs create by `create-graphs-for-different-experimental-configurations`.

4.5.3 *function* **create-graphs-for-different-experimental-configurations** *&key ...*

description This is very similar to `create-graphs-for-different-experimental-conditions`, with the difference that it does not require to create separate classes for each experimental condition. Instead, the same experiment class is run multiple times with different configurations.

example This example runs the `naming-game` experiment class 5 times, each time with a different probability for the invention of words:

```
(create-graphs-for-different-experimental-configurations
   :asdf-system "babel-demo"
   :package "babel-demo"
   :experiment-class "naming-game"
   :configurations
     '(((babel-demo::probability-for-word-invention .  1.0))
       ((babel-demo::probability-for-word-invention .  0.5))
       ((babel-demo::probability-for-word-invention .  0.25))
       ((babel-demo::probability-for-word-invention .  0.125))
       ((babel-demo::probability-for-word-invention .  0.05)))
   :captions '("1.0" "0.5" "0.25" "0.125" "0.05")
   :number-of-interactions 2000
   :number-of-series 10
   :data-recorders '("babel-demo::record-communicative-success"
     "babel-demo::record-average-number-of-words")
   :average-data '(t t)
   :parameters-for-graphic-generators
   '((:x-label "number of interactions"
      :y1-label "communicative success"
      :error-bars t :y1-max 1 :y1-min 0 :draw-y1-grid t
      :graphic-type "pdf" :key-location "right bottom"
      :file-name
        (babel-pathname
          :name "success-vs-invention-probabilities" :type "pdf"
          :directory '("experiments" "babel-demo" "graphs")))
     (:x-label "number of interactions" :y1-label "lexicon-size"
      :error-bars t :y1-min 0 :draw-y1-grid t
      :graphic-type "pdf" :key-location "right bottom"
      :file-name
        (babel-pathname
          :name "lexicon-size-vs-invention-probabilities" :type "pdf"
          :directory '("experiments" "babel-demo" "graphs")))))
```

The resulting-graphs are shown in figure 4.4.

`:asdf-system :package :number-of-interactions :number-of-series :data-recorders :average-data :parameters-for-graphic-generators`	The same as in `create-graphs-for-different-experimental-conditions` above.
`:experiment-class`	The class of the experiment to run with different configurations.
`:configurations`	Lists of configurations for each run. A configuration is a list of configuration / value pairs and the client processes pass them to the experiment with `set-configuration`.
`:captions`	A graph caption for each of the configurations.

4.5.4 *function* **create-graphs-for-different-population-sizes** *&key ...*

description	This function shows the impact of varying population sizes on particular measures. In order to make the different runs comparable, the x-axis does not show the number of interactions, but the number of interactions that each agent played on average. When for example the population size is 10 and the number of series 1000, then each agent will have played 200 games until the end.

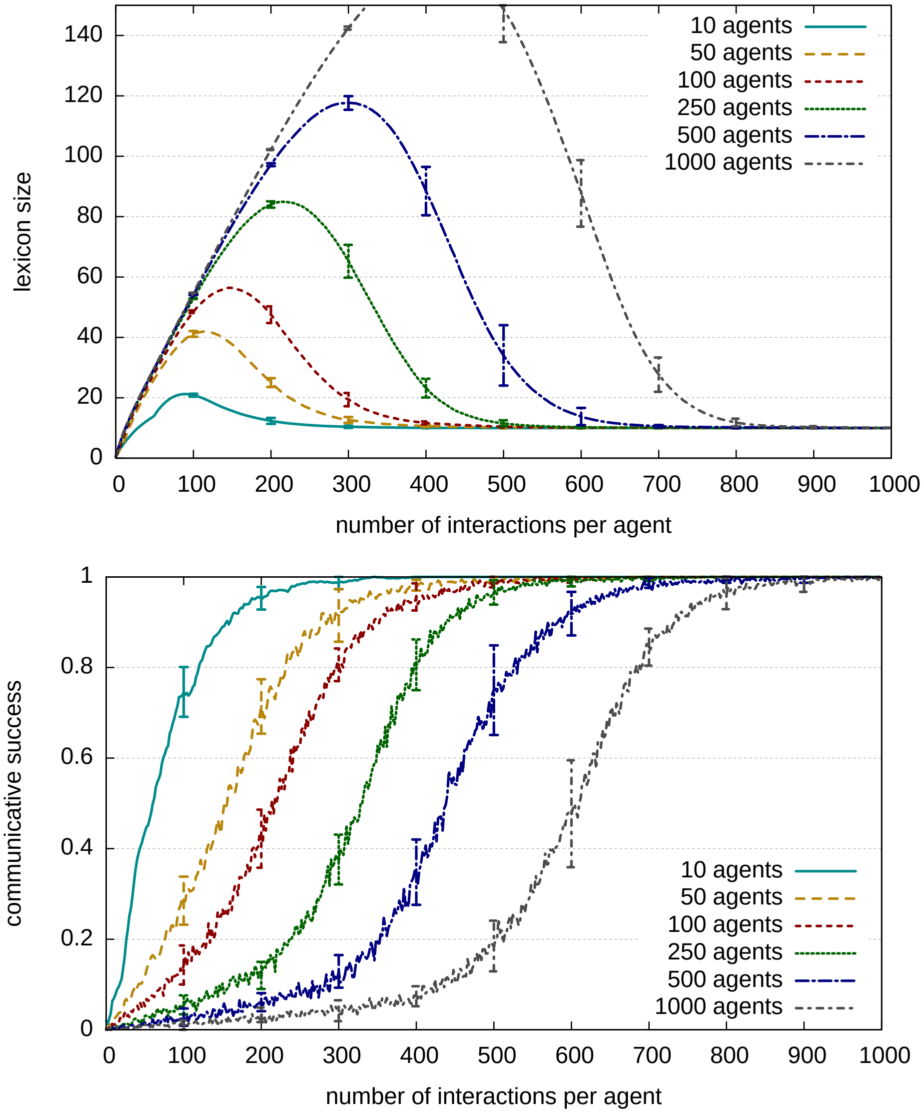

Figure 4.5: Examples for graphs create by `create-graphs-for-different-population-sizes`.

example	This example runs the `naming-game` experiment class with 6 population sizes:

```
(create-graphs-for-different-population-sizes
   :asdf-system "babel-demo"
   :package "babel-demo"
   :experiment-class "naming-game"
   :population-sizes '(10 50 100 250 500 1000)
   :number-of-interactions-per-agent 1000
   :number-of-series 10
   :data-recorders '("babel-demo::record-communicative-success"
                     "babel-demo::record-average-number-of-words")
   :average-data '(t t)
   :parameters-for-graphic-generators
   '((:key-location "bottom right"
      :x-label "number of interactions per agent"
      :y1-label "communicative success" :error-bars t
      :y1-min 0.0 :y1-max 1 :draw-y1-grid t
      :graphic-type "pdf" :colored t
      :file-name
        (babel-pathname
          :name "success-vs-population-size" :type "pdf"
          :directory '("experiments" "babel-demo" "graphs")))
     (:key-location "top right"
      :x-label "number of interactions per agent"
      :y1-label "lexicon size" :error-bars t  :y1-min 0
      :y1-max 150 :draw-y1-grid t
      :graphic-type "pdf" :colored t
      :file-name
        (babel-pathname
          :name "lexicon-size-vs-population-size" :type "pdf"
          :directory '("experiments" "babel-demo" "graphs")))
     ))
```

The resulting-graphs are shown in figure **??**.

`:asdf-system`
`:package`
`:experiment-class`
`:number-of-series`
`:data-recorders`
`:average-data`
`:parameters-for-graphic-generators`
	See above.

`:population-sizes`	A list of different population sizes. This will be used in the client processes to do `(set-configuration experiment 'population-size xxx)`, so you will have to create populations depending on `'population-size`.
`:number-of-interactions-per-agent`	How many interactions to run per agent. If for example the population size is 1000 and the `:number-of-interactions-per-agent` 1000, then 500000 interactions will be run by each client process.

5 Learning

Learning is deeply entrenched into the Babel framework. Learning in Babel cannot be only inductive learning since the agents also have to invent or change new items for their (linguistic) inventory to be gradually built up. We cannot assume (like is often done in machine learning algorithms) that there is a pre-given input set from which the agents can learn. In other words we do not assume that there is a teacher. All the input an agent ever gets is the world and output from other agents. This requires a constructivist approach to learning. To meet these requirements we have split up learning into *diagnosing* and *repairing*. In section 5.1 we present the definitions for the base classes including diagnostic, problem and repair-strategy. Sections 5.2 and 5.3 present the two instantiations of this base-framework that we have provided by default in Babel. These are the process level learning mechanisms and the agent level learning mechanisms. We use the term learning mechanism to signify both diagnostic and repair-strategy. Section 5.4 shows a detailed example on how to write a diagnostic, with a corresponding problem and repair-strategy.

I suggest for everybody who intends to write an experiment to read the complete chapter carefully.

5.1 Base classes

The classes in this section are abstract base classes of the learning mechanisms. Different levels of learning mechanisms subclass from them and add further semantics. All these classes can be found in the file "learning-mechanisms.lisp" in the experiment-framework.

5.1.1 *class* **diagnostic**

description	The base class for all diagnostics.

`situations`

```
(learning-situations :type list :reader learning-situations :initform nil
                     :initarg :learning-situations)
```

A situation narrows down the point of execution of a diagnostic. The kinds of learning-situations depends on the level the diagnostic is operating on. It is a list of symbols.

5.1.2 *monitor event* **diagnostic-started** *(diagnostic diagnostic)*

description	Triggered when a diagnostic starts running.
`diagnostic`	The diagnostic that is run.

5.1.3 *class* **problem**

---

description

Represents a problem. A problem is created by diagnostics to signal a failure or some inefficiency and contains all the information necessary to deal with the problem.

slot `issued-after`

```
(issued-after :type symbol
              :accessor issued-after
              :initform nil
              :initarg :issued-after)
```

This is a symbol representing when this problem was signaled. When created by a process-diagnostic this is the name of the process.

slot `repaired-by`

```
(repaired-by :type t :accessor repaired-by
                     :initform nil
                     :initarg :repaired-by)
```

This slot is automatically set to the repair-strategy that repaired it. When it's nil it means it's still unrepaired

5.1.4 *monitor event* **diagnostic-returned-problems** *(diagnostic diagnostic) (problems list)*

description

Triggered after a diagnostic was run and only if it returned at least one problem.

`diagnostic`

The diagnostic that was run.

`problem`

The list of problems that was returned.

5.1.5 *class* **repair-strategy**

description

Base class for all repair strategies. Repair strategies are able to deal with a set of problems.

slot `triggered-by-problems`

```
(triggered-by-problems :type list :reader triggered-by-problems
                       :initform nil
                       :initarg :triggered-by-problems)
```

A list of problems (class names of problems) that the repair strategy might be able to fix.

slot `learning-situations`

```
(learning-situations :type list :reader learning-situations :initform nil
                     :initarg situations)
```

situation narrows down the point of execution of a repair-strategy. The kinds of learning-situations depends on the level the repair-strategy is operating on. It is a list of symbols.

slot `success-score`

```
(success-score :type number :accessor success-score
               :initform 1.0 :initarg :success)
```

Resembles how successful the repair-strategy is. The higher the better. This will be used to sort when multiple repair-strategies can be triggered at the same time.

5.1.6 *monitor event* **repairing-started** *(repair-strategy repair-strategy) (problem problem))*

description	When a repair strategy is called.
`repair-strategy`	The repair strategy that is called.
`problem`	The problem on which it is called.

5.1.7 *monitor event* **repairing-finished** *(repair-strategt repair-strategy) (repaired boolean))*

description	After a repair strategy was run.
`repair-strategy`	The repair strategy that was run.
`repaired`	Whether it was able to repair or not.

5.1.8 *class* **object-with-learning-mechanisms**

description: A helper class that provides derived classes with learning mechanisms. At the moment of writing the classes that derive from this are an experiment [4.2.1.1], an agent [4.1.2.1] and a task [6.1.1.1].

slot `diagnostics`

```
(diagnostics :type list :accessor diagnostics
             :initarg :diagnostics :initform nil)
```

A list of diagnostics.

slot `repair-strategies`

```
(repair-strategies :type list :accessor repair-strategies
                   :initarg repair-strategies :initform nil)
```

A list of repair strategies.

5.1.9 *generic function* **add-diagnostic** *object diagnostic*

description/ default implementation	Adds a diagnostic to `object`.
`object`	Anything that is derived from `object-with-learning-mechanisms`.
`diagnostic`	An instance of a diagnostic. If there is already an diagnostic of the same class as `diagnostic` in `object`, that diagnostic is replaced and you will get a warning.

example	`(add-diagnostic *experiment* (make-instance 'uncovered-meaning-diagnostic))`

5.1.10 *generic function* **delete-diagnostic** *object diagnostic*

description/ default implementation	Deletes a diagnostic from `object`.
`object`	Anything that is derived from `object-with-learning-mechanisms`.
`diagnostic`	A diagnostic of `object`.

5.1.11 *generic function* **add-repair-strategy** *object repair-strategy*

description/ default implementation	Adds a repair strategy to `object`.
`object`	Anything that is derived from `object-with-learning-mechanisms`.
`repair-strategy`	An instance of a repair strategy.

5.1.12 *generic function* **delete-repair-strategy** *object repair-strategy*

description/ default implementation	Deletes a repair strategy from `object`.
`object`	Anything that is derived from `object-with-learning-mechanisms`.
`repair-strategy`	A repair strategy of `object`.

5.1.13 *monitor* **trace-learning** *trace-monitor*

description	Prints information on detected and repaired problems.

5.1.14 *monitor* **trace-learning-verbose** *trace-monitor*

description	In addition to the stuff printed by monitor trace-learning, it also prints which learning mechanisms are run and changes of an agent's inventories (for example when rules are added or modified).

5.2 Process level learning

In Babel one can learn (by default) both at the level of processes (see chapter **??** if you don't know what processes are) and at the level of an agent. In this section we focus on the process level learning mechanisms.

A process-diagnostic can be run after any given process. It can report a problem if it detects one. After every process there is a check for new problems and the process level repair strategies get a chance to fix them. In case of a successful repair the repaired-by slot of the process get's set to the repair-strategy and there might be a restart to a previous process. This will become more clear when you get read about the classes and their methods further down. All of this can be found in the file "process-learning-mechanisms.lisp" in tasks-and-processes.

5.2.1 *class* **process-diagnostic** *diagnostic*

description	A diagnostic that is triggered after the execution of a process. Examples of possible learning-situations are production, re-entrance, interpretation, production.
slot `trigger-processes`	`(trigger-processes :type list :reader trigger-processes :initform nil)` The names of processes after which this diagnostic should be triggered.

This means the exact point of execution of a process-diagnostic is a combination of the learning-situations and the trigger-processes. For example if the learning-situations are '(production) and the trigger-processes are '(apply-lex-stem) the diagnostic will only be called during production and after the lex-stems were applied.

To run a process-diagnostic one has to implement a diagnose-process method. If you define a process-diagnostic without supplying an implementation for this method an error will be thrown. Diagnose-process should return either one problem, a list of problems or nil. If this is not the case an error will be thrown. In case one or more problems are returned they are automatically added to the problems of the task that is currently running. When nil is returned it means that the diagnostic did not diagnose anything.

5.2.2 *generic function* **diagnose-process** *process-diagnostic task process*

description/ default implementation	Diagnose-process is called after running a process and handling it's process-results. It's exact call-location depends on the combination of it's trigger-processes and it's learning-situations. e.g. If trigger-process is apply-lex-stem and situation is production this method will only be executed after apply-lex-stem during production of the speaker and nowhere else.
`process-diagnostic`	The diagnostic you want to specialise on.
`task`	You will need this task for diagnosing a potential problem.
`process`	A process is just a symbol but it might be necessary if the diagnostic can be triggered after different processes. Since we provide you with the process you know immediatly in which case you are. You can even specialise on it.

5.2.3 *class* **task-problem** *problem*

description	A problem is always related to a task. Therefore you should always derive from this and not from the base-class problem.

slot `task`

```
(task
    :type task
    :initform (error "When creating a task-problem you have to supply a :task")
    :initarg :task
    :accessor task)
```

The task this problem is about.

5.2.4 *class* **process-repair-strategy** *repair-strategy*

description	A process-repair-strategy can be triggered on a problem. Process repair-strategies are checked in between every process.

Every process-repair-strategy should have a repair-process method specialised on it. Repair-process can return two values. The first a boolean whether the repair succeeded or not. Second a process (which is a symbol) to which the task must be restarted. If this is nil it will just continue. I would like to stress that one should be very careful in setting the first boolean to true and restarting. Because if the problem during the restart gets signaled again the repair will be triggered again and one potentially finds himself in an infinite loop. So only return true (with a restart) when you are very certain that all necessary modifications have been made so that the problem will not get diagnosed anew.

5.2.5 *generic function* **repair-process** *repair-strategy problem task process*

description/ default implementation	In between every process the problems will be checked and if possible a correct repair-process will be executed. It can return two values. The first a boolean whether the repair succeeded or not. Second a process (which is a symbol) to which the task must be restarted. If this is nil it will just continue.
`repair-strategy`	The repair-strategy you want to specialise on.
`task`	The current task. You will probably also need it for repairing.
`problem`	The problem that triggers the repair-strategy. You probably need to specialise on this too. A problem can contain many usefull slots containing information already gathered during diagnosing.
`process`	A process is just a symbol but it might be necessary if the diagnostic can be triggered after different processes. Since we provide you with the process you know immediatly in which case you are. You can even specialise on it.

5.3 Agent level learning

Sometimes it is impossible to diagnose or repair something in between processes. One reason is that at the process level you do not have all the information necessary to perform the diagnosis. For example when you need re-entrance information and compare this to production. Sometimes it is possible to diagnose something after a given process but can only repair it later e.g. after receiving pointing information.

One of the nicest features of the learning framework is that problems are "level-independent". A problem diagnosed by a process level diagnostic can be repaired by an agent level repair-strategy. This is possible because all the problems from the (best) task get copied to the agent when the task has finished. Actually it runs deeper, there is no direct link between a diagnostic and a repair-strategy. Their connection is only indirect by the use of problems.

We will start by presenting the most important classes. All the information presented here can be found in "agent-learning-mechanisms" in experiment-framework.

5.3.1 *class* **agent-diagnostic** *diagnostic*

description	A diagnostic that is triggered after run-agent.

For every agent-diagnostic one has to supply a diagnose-agent method.

5.3.2 *generic function* **diagnose-agent** *diagnostic agent-interaction-point agent world*

description/ default implementation	After run-agent diagnose-agent will be called for every agent-diagnostic. They have to return either one problem, a list of problems or nil. The problems will be pushed onto the problems of the agent automatically.
`diagnostic`	The diagnostic you want to specialise on.
`agent-interaction-point`	This is a more specific name for a learning situation at the agent level. It is a symbol and can be used to specialise or just to check what the interaction-point is.
`agent`	The agent that is currently running.
`world`	This is one of the major differences with a process-diagnostic that one has access to the world at this level.

5.3.3 *class* **agent-repair-strategy** *repair-strategy*

description	These repair strategies are executed after `run-agent`. They try to repair problems in the agent, which could also be problems created by lower-level diagnostics.

Every agent-repair-strategy should supply a repair-agent method. A repair-agent-method can return two values. The first a boolean whether the repair was successful or not. Second can be anything which is guaranteed to be put into the "rerun-data" slot of the agent. The second value only matters when the first is non nil and consists of rerun-data. When it is not nil it signifies a restart of run-agent. This second value will also be stored automatically in the “rerun-data” slot of the agent. This allows one to change the behaviour during a rerun. For example one could skip conceptualisation.

5.3.4 *generic function* **repair-agent** *repair-strategy agent-interaction-point problem agent world*

description/ default implementation	repair-agent is called after run-agent. It might however also repair problems created by lower-level diagnostics. It can return two values. The first one a boolean signifying whether the repair was successful. The second rerun-data that is automatically stored in the rerun-data slot of the agent. When the rerun-data is non-nil a restart will be initiated.”
`repair-strategy`	The repair-strategy you want to specialise on.
`agent-interaction-point`	This is a more specific name for a learning situation at the agent level. It is a symbol and can be used to specialise or just to check what the interaction-point is.
`problem`	The problem that triggers the repair-strategy. You probably need to specialise on this too. A problem can contain many usefull slots containing information already gathered during diagnosing.
`agent`	You have access to the agent during repairing.
`world`	You have access to the world during repairing.

5.3.5 *class* **rerun-data**

description	An empty abstract class that can be used to derive from when creating objects to return as second value in repair-agent. Rerun-data-with-restored-task which is used in the language-game templates is derived from this.

5.3.6 *class* **rerun-data-with-restored-task** *rerun-data*

slot `trigger-processes`	`(trigger-processes :type list :reader trigger-processes :initform nil)` The names of processes after which this diagnostic should be triggered.

5.4 Detailed example

We will start with an example for writing process level learning mechanisms. If you are not yet acquanted with the cookie-baking example in section 6.3 from chapter **??** then I suggest you read that first because the current example builds further on that one.

For clarity reasons, here is what we had so far.

```
(defclass cookie-baking-agent (agent-with-tasks)
```

```
  ()
  (:documentation "An agent capable of making delicious cookies."))

(defmethod initialize-instance :after ((agent cookie-baking-agent) &key)
  (add-data-field agent 'available-ingredients nil))

(defclass simple-cookie-baking-task (task)
  ()
  (:documentation "An implementation of a task for baking simple cookies."))

(defmethod initialize-instance :around ((task simple-cookie-baking-task) &key &allow-other-keys)
  (call-next-method)

  (add-data-field task 'used-ingredients nil)
  (add-data-field task 'missing-ingredients nil)
  (add-data-field task 'cookies nil)

  (add-process task 'find-all-ingredients nil)
  (add-process task 'make-cookies '(find-all-ingredients))
  (add-process task 'bake-cookies '(make-cookies)))

(defmethod run-process ((task simple-cookie-baking-task)
                        (process (eql 'find-all-ingredients)))
  (if (subsetp '(chocolate flour)  (get-data task 'available-ingredients))
     (list (make-process-result :succeeded t :confidence 1.0
                 :data (list (cons 'ingredients (chocolate flour)))))
     (list (make-process-result :succeeded nil :confidence 0.0
                 :data (list
                           (cons 'ingredients
                                 (intersection '(chocolate flour)
                                               (get-data task 'available-ingredients))))))))

(defmethod run-process ((task simple-cookie-baking-task)
                        (process (eql 'make-cookies)))
  ...
  (list (make-process-result ...)))

(defmethod run-process ((task simple-cookie-baking-task)
                        (process (eql 'bake-cookies)))
  ...
  (list (make-process-result ...)))

(defmethod goal-achieved ((task simple-cookie-baking-task))
   (and (find 'bake-cookies (finished-processes task))
        (succeeded (get-process-result task 'bake-cookies))))
```

What would happen if an agent runs out of some ingredients. This would mean that every cookie-baking task would fail since find-all-ingredients would fail.

To solve this issue we define a process-diagnostic that is able to detect that ingredients are missing.

```
(defclass detect-missing-ingredients (process-diagnostic)
```

```
  ()
  (:documentation "After running find-all-ingredients this
  diagnostics checks whether there where ingredients missing."))

(defmethod initialize-instance :after ((diagnostic detect-missing-ingredients) &key)
  (setf (slot-value diagnostic 'trigger-processes) '(find-all-ingredients))
  (setf (slot-value diagnostic 'learning-situations) '(baking)))
```

Note the after method that sets the learning-situations and at this point more important the trigger-processes. We also create a problem that this diagnostic can create in case of a shortage of ingredients.

```
(defclass missing-ingredients-problem (task-problem)
  ((missing-ingredients :documentation "A list of the missing ingredients."
      :initform (error "Please supply :missing-ingredients.")
      :initarg :missing-ingredients :accessor missing-ingredients :type list))
  (:documentation "This problem is created when there are missing ingredients."))
```

It has one slot that can be used to store the missing ingredients. In this way the repair-strategy does not have to look for them again but can just acces them from this problem.

Now it's time to supply the diagnose-agent method which will do the diagnosing.

```
(defmethod diagnose-process ((diagnostic detect-missing-ingredients)
     (task task) (process symbol))
  (let ((process-result (get-process-result task 'find-all-ingredients)))
    (when (and (not (pr-succeeded process-result))
       (field? (pr-data process-result) 'missing-ingredients))
      (make-instance 'missing-ingredients-problem
     :missing-ingredients (get-data process-result 'missing-ingredients)
     :task task))))
```

This is all that is necessary for creating a diagnostic that will successfully diagnose missing ingredients. Off course we need to supply a repair-strategy that can handle this problem by buying the missing ingredients.

```
(defclass buy-missing-ingredients (process-repair-strategy)
  ()
  (:documentation "This repair strategy will add some more
  ingredients to the available ingredients."))

(defmethod initialize-instance
    :after ((repair-strategy buy-missing-ingredients) &key)
  (setf (slot-value repair-strategy 'triggered-by-problems) '(missing-ingredients-problem))
  (setf (slot-value repair-strategy 'learning-situations) '(baking)))
```

The after method makes sure that the repair-strategy knows which problems it might be able to repair. It has the same learning-situations so will trigger right after the diagnostic. The repair-agent method can be implemented as follows:

```
(defmethod repair-process ((rs buy-missing-ingredients)
```

```
   (problem missing-ingredients-problem) (task task)
   (process symbol))
  (loop for (ingredient amount) in (get-data (agent-data task) 'available-ingredients)
       when (find ingredient (missing-ingredients problem) :test #'equal)
       do (nsubst (list ingredient 5)
  (list ingredient 0)
  (get-data (agent-data task) 'available-ingredients)
  :test #'tree-equal))
  (when (every #'(lambda (item) (> (second item) 0))
                   (get-data (agent-data task) 'available-ingredients))
    (values t 'find-all-ingredients)))
```

This will search for ingredients for which we have zero in stock and put 5 new there. It will restart at the beginning of find-all-ingredients but only when it is certain that it has indeed supplied all the necessary ingredients.

6 Tasks and Processes

Most of this chapter is not needed when writing experiments. When using the language game templates you will only get confronted with tasks when writing learning mechanisms.

For those that would like to read the minimum and are not interested in writing their own tasks we advise section 6.1.1.1 and when you're interested in writing process learning mechanisms it is strongly advised to read chapter 5 carefully and since you will be accessing information about the processes it is also recommended to read section 6.2.

Section 6.3 clearly shows an example of how to create a task and knowing how to write your own will most certainly help you in understanding the default tasks.

For those who are interested how tasks and processes are run behind the scenes there is section 6.1.2. If you are writing advanced learning mechanisms it might also be a good read.

6.1 Tasks, task-processors and task-results

Tasks are used for maintaining and running processes. They are the primary interface to tasks and processes. From the moment you want more control over your experiments then the default behaviour you will have to deal with tasks. In section 6.1.1.1 class task is explained in full detail.

6.1.1 Task

6.1.1.1 *class* **task** *object-with-learning-mechanisms*

description A "task" inherits from object-with-learning-mechanisms. Besides the slots it inherits from its superclasses it contains some new slots that are specific to a task.

slot `(id :type symbol :reader id)`

A symbol that is generated automatically during instantiation by incrementing the id-counter and prepending it with 'TASK. Therefore id's are of form "TASK-X".

slot `(data :type blackboard :accessor data :initarg :data)`

Data that is local to the task and exchanged between processes.

slot `(agent-data :type blackboard :accessor agent-data :initarg :agent-data)`

(A copy of) the persistent data (inventories) of the agent.

slot `(processes :type list :initform nil :reader processes)`

An alist of processes and their dependencies that are used by this task. A list of conses (a . (b c)): a depends on b and c.

slot `(finished-processes :type list :initform nil :accessor finished-processes)`

The processes that have been run.

slot `(process-results :type list :initform nil :accessor process-results)`

An alist of all finished processes and their process-result. When a process finished it returns a process-result. This result is always (irrespective of it being successful or not) added to the process-results.

slot `(problems :type list :initform nil :initarg :problems :accessor problems)`

All the problems that were reported by diagnostics operating on a task level or lower. (e.g. the process-diagnostics)

slot
```
(data-states :type list :initform nil
             :initarg :data-states :accessor data-states)
```

We keep a copy of the data-slot at the beginning of every process. This allows for very easy, fast and non-ambiguous restoring of a task. Although it comes at some copying cost during task execution. It is an alist of (process . data).

slot `(agent :type list :initform nil :initarg :agent :accessor agent)`

A pointer to the agent which created this task (if applicable). Added this slot for monitoring purposes only.

slot
```
(configuration :type configuration
               :initarg :configuration :accessor configuration)
```

For configuring the task.

First we present the generic functions that you will probably need when creating your own task or writing learning mechanisms.

6.1.1.2 *generic function* **get-process-result** *task process*

description/ default implementation

Returns the process-result for the given process. It errors when it cannot find a process-result for the given process.

6.1.1.3 *generic function* **run-process** *task process*

description/ default implementation

This is the method that has to be implemented for every process. It is in a sense what defines the process. It should return a list of process-results. Even when a process fails it should return a list containing one process-result.

6.1.1.4 *generic function* **goal-achieved** *task*

description/ default implementation	Returns t if the goal of a task was achieved. It is a very important method because a task can only be successful if it passes this test. When it does it also completely stops the running of any other processes from the task because it has achieved what it needed to.

6.1.1.5 *generic function* **finished-processes** *task*

description/ default implementation	Returns a list of all processes that have been run irrespective of them being successful.

6.1.1.6 *generic function* **add-process** *task process dependencies*

description/ default implementation	Adds a process and it's dependencies to a task.

6.1.1.7 *generic function* **delete-process** *task process*

description/ default implementation	Deletes a process and it's dependencies from a task.

6.1.1.8 *generic function* **run-task** *task*

description/ default implementation	Runs the given task. It returns a task-result-collection since running a task could spawn many new tasks in case of ambiguity.

The following generic functions are more internal to the execution of tasks and processes and you will most probably not need them unless you are interested in changing how a task is run.

6.1.1.9 *generic function* **get-all-process-dependencies** *task process &optional result*

description/ default implementation	Returns a list of all the processes that *have to be finished before* this process can be run. This is a recursive (and deeper) variant of get-process-depencies.

6.1.1.10 *generic function* **get-all-dependent-processes** *task process &optional result*

description/ default implementation	Returns a list of all processes that *cannot* be run before the given process is run. So all the processes that are directly or indirectly dependent on the given process.

6.1.1.11 *generic function* **dependencies-solved?** *task process*

description/ default implementation	Returns true if all the dependencies of the given process are solved.

6.1.1.12 *generic function* **get-processes-without-dependencies** *task*

description/ default implementation	Returns a list of processes without their dependencies. This is just a list of all the processes the task knows.

6.1.1.13 *generic function* **get-process-dependencies** *task process*

description/ default implementation	Returns a list of processes on which the given process depends. The list contains only the direct dependencies. e.g. If 'a' depends on 'b' and 'b' depends on 'c' and you ask get-process-dependencies for 'a' it will only give you 'b' and not 'c'. If you want the complete list of dependencies use get-all-process-dependencies.

6.1.1.14 *generic function* **add-process-result** *task process process-result*

description/ default implementation	Add the process-result to the process-results of the object.

6.1.2 Behind the scenes: Running of a Task

Running all the processes in a task requires a lot of bookkeeping. For example one needs to keep track of all the processes that have been run, after every process check whether dependencies of some processes have been met and queue them so they can be run next. Since we did not want to clutter the class task, we created a new structure *task-processor* that contains one task, namely the task that is being run and has some extra slots such as a process-queue for bookkeeping. Normally you should never have to create a task-processor yourself. This is all done automatically behind the scenes. Moreover chances are very small you will ever interface with a task-processor since when a task has finished a task-result (see section 6.1.2.4) is created and the task-processor gets collected in the garbage.

6.1.2.1 *structure* **task-processor** *node*

description	A task-processor contains a task and other information needed for running the processes in this task.

slot

```
(task :type task :accessor tp-task :initarg :task
 :initform (error "Please provide a task
                   when creating a task-processor."))
```

The task this task-processor task is processing.

slot

```
(confidence :type float :accessor tp-confidence
            :initarg :confidence :initform 0.0)
```

This is used during running tasks for determining which should be run next. So tasks are run by priority on their confidence. And also in the end the bast-task is the one with the highest confidence. The confidence of a task is a function of the confidences of the processes it has run.

slot

```
(process-queue :type queue :accessor process-queue
  :initarg :process-queue :initform (make-instance 'queue))
```

The process-queue is used for keeping track of the processes when running them.

6.1.2.2 *generic function* **restart-task** *task-processor process*

description/ default implementation	Restart a task at a specific process. Restores the process-queue of the task-processor and rewrites previous process results to the black board.

6.1.2.3 *generic function* **run-processes** *task-processor*

description/ default implementation	Try running all processes of the task inside the task-processor.

There are three different possibilities for a task to stop.

1. First (goal-achieved task) could return true. When this is the case the task has achieved it's goal and stops it's execution.
2. Second when a task has no more processes to run (because it has run them all, or because some dependencies are not met) and the goal has not been achieved yet the task simply fails.
3. Third the task could also "split" into multiple new tasks. This only happens when run-process returns more then one process-result. When this is the case the original task also stops and the newly created tasks take over. In a sense the task not really stops, it just hands over responsibility to it's children.

In the first two of these cases an object of structure "task-result" is created. Normally you never have to create a task-result yourself. You will however get in contact with task-results when writing learning mechanisms. In section 6.1.2.4 you find all the details regarding this class.

6.1.2.4 *structure* **task-result** *node*

description	A task-result is created when a task has finished running. It contains the finished task and some extra slots to indicate whether it succeeded or not and a confidence. This means it has a contains-a relation with the task and not a is-a relation. It derives from node which means it can be used in a tree structure.

slot

```
(task :type task :initform (error "Please provide a task
                                  when creating a task-result.")
      :initarg :task :accessor tr-task)
```

This is the task the task-result is about. Since you only create a task-result when the task has finished you have to immediatly supply the task via the :initarg.

slot

```
(confidence :type float :initarg :confidence :accessor tr-confidence
            :initform (error "Please provide a confidence
                              value when creating a task-result"))
```

A value between 0 and 1. 1 meaning that you are very confident. In most cases this will be a function of the process-results of the task.

slot

```
(succeeded :type boolean :accessor tr-succeeded :initarg :succeeded
           :initform (error "Please provide a boolean value for
                     succeeded when creating a task-result"))
```

Indicates whether the goal of the task was achieved.

Some generic functions from section 6.1.1.1 are also specialised for a task-processor and a task-result. The call is just passed to task contained in the task-processor or task-result. These are:

- id
- processes
- process-results
- finished-processes

Since a task can split into multiple new tasks and these tasks can split again it is obvious that the running of one task cannot always return one task-result. It might be many different task-results, some of them succeeded others failed. We have captured this in a new structure named task-result-collection. It includes structure mtree which means that is also a tree. When writing learning mechanisms you will most probably have to interface with this class very often.

6.1.2.5 *structure* **task-result-collection** *mtree*

description

Contains task-results and task-processors structured as a tree. When running a task it is an instance of task-result-collection that is returned. In the most simple case (when there is no ambiguity) this structure will only contain one task-result. However when ambiguous tasks are run it will be a far more elaborate structure with different task-results.

slot

```
(succeeded-task-results :type list :accessor succeeded-task-results
        :initform nil)
```

All succeeded task-results. In many cases this will just contain one element.

slot

```
(failed-task-results :type list :accessor failed-task-results
     :initform nil)
```

All failed task-results.

6.1.2.6 *generic function* **best-task-result** *task-result-collection*

description/ default implementation	Returns task-result with the highest confidence. It first tries searching succeeded-task-results and if this is empty it tries failed-task-results.

6.1.2.7 *generic function* **best-task** *task-result-collection*

description/ default implementation	Returns the task inside the task-result with the highest confidence. It first tries searching succeeded-task-results and if this is empty it tries failed-task-results.

To conclude this section we present a simplified and incomplete version of how a task is run:

```
(defmethod run-task ((task task))
  (let ((task-result-collection (make-task-result-collection))
        (task-queue (make-instance ’queue))
        active-task-processor)
    (enqueue-by-priority task-queue (make-instance ’task-processor :task task) #’tp-confidence)
    (loop until (empty-queue? task-queue)
      do
      (setf active-task-processor (pop-front task-queue))
      (solve-process-dependencies active-task-processor)
      (cond ((goal-achieved (task active-task-processor))
             ;; the current task succeeded
             (add (make-task-result :task (tp-task active-task-processor)
                            :succeeded t)
                  task-result-collection))
            ((not (empty-queue? (tp-process-queue active-task-processor)))
             ;; there still are processes to run
             (let ((new-tasks (run-processes active-task-processor)))
               (enqueue-by-priority task-queue new-tasks #’tp-confidence)))
            (t ;; the current task failed
             (add (make-task-result :task (tp-task active-task-processor)
                                  :succeeded nil)
                   task-result-collection))))
    task-result-collection))
```

6.2 Processes and Process-results

Processes themselves are not modelled as classes since the only thing that defines them is what they do. A process is just a symbol like 'render or 'apply-con-rules but every such symbol should also have a run-process method that specializes on that symbol with an eql statement (also see 6.3 for an example). Since processes are not modelled they use the task they are part of as a blackboard to write their output data. When run-process finishes the data from the process-result is written to the task and the process-result itself is also fully stored in the task for in case one would like to investigate it for learning.

When a process is running (so inside the run-process of that process) three different scenario's are possible.

1. First everything goes well and the run-process just returns a list containing one process-result with it's succeeded-slot set to true.
2. Second something went wrong and the process cannot be run successfully. In the cookie baking example it could be that you run out of ingredients. In this case one returns also a list containing one process-result but with succeeded-slot set to nil.
3. Third it could be that there is some kind of ambiguity. e.g. The recipe states that one should add sugar but you don't know whether it's brown or white sugar. In this case run-process should return a list process-results with a process-result for every possibility.

A process-result therefore plays a very important role not only in running the task but also when diagnosing or repairing you will interface with the process-results quite often. All process-results are remembered in the slot process-results of the task.

6.2.1 *structure* **process-result**

description The output of the execution of a process.

slot

```
(data :accessor pr-data :type list :initarg :data :initform nil)
```

The data you wish to return to the task. It's an alist containig pairs of (datafield . data).

slot

```
(confidence :type float :accessor pr-confidence
            :initarg :confidence :initform 0.0)
```

How confident the process is of this result.

slot

```
(succeeded :type boolean :accessor pr-succeeded
           :initarg :succeeded :initform nil)
```

Whether depending procecces should be triggered.

6.2.2 *generic function* **handle-process-result** *task-processor process process-result*

description/ default implementation Checks a process-result and writes all necessary changes to the task-processor (including the task it contains).

6.2.3 *generic function* **handle-process-results** *task-processor process list*

description/ default implementation Handles a list of process-results. This may change the task-processor but it may also spawn new tasks.

6.3 Implementing your own task

Implementing your own task with it's own processes is very easy.

Assume you wish to create a task for baking cookies. This task contains three processes:

1. find-all-ingredients
2. make-cookies
3. bake-cookies

Although this is not entirely necessary we first create an agent because it makes more sense.

```
(defclass cookie-baking-agent (agent-with-tasks)
  ()
  (:documentation "An agent capable of making delicious cookies."))

(defmethod initialize-instance :after ((agent cookie-baking-agent) &key)
  (add-data-field agent 'available-ingredients nil))
```

Now we are ready to create a new class that derives from task or a subclass of task.

```
(defclass simple-cookie-baking-task (task)
  ()
  (:documentation "An implementation of a task for baking simple cookies."))
```

Remember that a task already derives from object-with-learning-mechanisms. So although this new task looks empty it is not. The next step is to let the task know about it's processes and their dependencies. It is obvious that one cannot bake cookies before one has made the cookies and one cannot make cookies before one has found all necessary ingredients. So we have a linear dependency between the three processes. We do this as follows:

```
(defmethod initialize-instance
           :around ((task simple-cookie-baking-task) &key &allow-other-keys)
  (call-next-method)

  (add-data-field task 'used-ingredients nil)
  (add-data-field task 'missing-ingredients nil)
  (add-data-field task 'cookies nil)

  (add-process task 'find-all-ingredients nil)
  (add-process task 'make-cookies '(find-all-ingredients))
  (add-process task 'bake-cookies '(make-cookies)))
```

The next thing is to implement run-process methods for all three processes. The most important thing to note is that they specialize on process with eql and that they always return a list of process-results.

```
(defmethod run-process ((task simple-cookie-baking-task)
                        (process (eql 'find-all-ingredients)))
  (if (subsetp '(chocolate flour)  (get-data task 'available-ingredients))
     (list (make-process-result :succeeded t :confidence 1.0
                 :data (list (cons 'ingredients (chocolate flour)))))
     (list (make-process-result :succeeded nil :confidence 0.0
                 :data (list (cons 'ingredients
                                   (intersection '(chocolate flour)
                                                  (get-data task 'available-ingredients)))))))
```

```
(defmethod run-process ((task simple-cookie-baking-task)
                        (process (eql 'make-cookies)))
  ...
  (list (make-process-result ...)))

(defmethod run-process ((task simple-cookie-baking-task)
                        (process (eql 'bake-cookies)))
  ...
  (list (make-process-result ...)))
```

There is only one thing left to do now, which is to provide a test that allows the task to know it has achieved it's goal. This is done by implementing the goal-achieved method.

```
(defmethod goal-achieved ((task simple-cookie-baking-task))
   (and (find 'bake-cookies (finished-processes task))
        (succeeded (get-process-result task 'bake-cookies))))
```

A fully working implementation can be found in the file simple-cookie-baking-task.lisp which can be found under experiment cookie-experiment in folder experiments.

6.4 Process Learning Mechanisms

Since learning is crucially important we have dedicated a chapter to accomodate all there is to know about learning in the Babel framework. We refer you to chapter 5. In that chapter the above example will also be further expanded with learning mechanisms.

7 The FCG core

7.1 General Information

Fluid Construction Grammar grew out of a larger body of research studying the origins and evolution of language. In earlier work, it has already been shown how large populations of autonomous agents can develop shared lexicons and ontologies (Steels, 2001). FCG is part of attempts to carry this research into the domain of grammar. More specifically, how the grounding of experiments in the real world plays a role in grammar construction and what kind of grammatical framework would be most suited for this purpose.

A promising direction has been taken under the influence of Construction Grammar (Goldberg, 1995) and its associated learning framework, Constructivist Learning (Tomasello, 2003). Many of the principles underlying these theories have been implemented in FCG by building on existing techniques from computational linguistics (such as unification and feature structures) and using them in a novel way. In this section, we will explain how FCG relates to other construction grammars and describe additional principles underlying the formalism.

7.1.1 The Role of Construction Grammar

As the name indicates, Fluid Construction Grammar shares a lot of common ground with other construction grammar theories. The common aspects of construction grammars, dubbed 'Vanilla Construction Grammar' by (Croft, 2005), can therefore be considered as essential foundations of FCG. The most important are:

1. Lexical and grammatical inventories consist of constructions, that is, pairings of form and meaning. Cognitive linguists have generalized this notion of a construction to representations that range from purely syntactic units to complex symbolic form-meaning associations. This allows for a uniform way of representing all grammatical knowledge, better known as the syntax-lexicon continuum. This view is very different from the sharp distinction between lexicon and grammar as proposed by many other syntactic theories;
2. A speaker's knowledge about her language is seen as a structured inventory or taxonomy of constructions. In FCG this structure is represented by the fact that – even though all the linguistic knowledge is represented in exactly the same way – some constructions may differ in their function when licensing a construct. This enables us to label the different levels of structure to facilitate linguistic analysis.

3. Construction Grammar is a *usage-based* model of language (Langacker, 2000). Usage-based models acknowledge the importance of linguistic experience and frequency not only for linguistic performance, but also for linguistic 'competence'. In our model, linguistic knowledge is stored in the form of constructions (pairings of form and meaning) that each have a success score. This success score is based on the communicative success that the agents have when applying these constructions during interactions. The result is a dynamic, emergent linguistic system that is continually shaped by linguistic usage (see also Hopper (1987)).

Fluid Construction Grammar has been implemented as a unification-based formalism, which has also been proposed for other types of construction grammar (Kay and Fillmore, 1999). Unification is one of the core mechanisms of FCG together with the merge operation.

7.1.2 Additional Design Principles

Because Fluid Construction Grammar supports 'Vanilla Construction Grammar', it can be used as a theory-independent tool for linguistic research. This enables the experimenter to test different alternative hypotheses and shape the formalism to theory-specific needs. However, FCG does have some additional properties that might place it closer to particular branches of construction grammars or cognitive linguistics.

First of all, FCG stresses the uniform representation of linguistic knowledge by having the exact same grammatical inventory for both parsing and production. In other words, every construction has to be *bi-directional*. This bi-directionality is a tough but necessary challenge because we are interested in agents capable of both parsing and production.

7.1.3 Variables and Bindings

Variables are of crucial importance in the FCG formalism since they allow the linking of different constructions. Variables are denoted with a question mark in front and they are treated as logic variables that get bound by matching processes. For example, the following expression could be part of the sentence *Jill gives Jack a block*:

```
give(?ev), give-1(?ev, ?obj1),
give-2(?ev, ?obj2), give-3(?ev, ?obj3)
```

This matches with the world model presented above to yield the following bindings:

```
((?ev .  ev1) (?obj1 .  obj1) (?obj2 .  obj2) (?obj3 .  obj3))
```

7.1.4 Coupled Feature Structures

The basic information structure of FCG is a feature structure with a set of units, associated slots for each unit, and constraints on the slots. A feature structure can look like this:

```
((unit-1 (meaning ((John John)))))
```

This example shows a feature structure with only one unit with name (`unit-1`). The associated slot `meaning` is filled by a list of meaning predicates. In FCG, feature structures are always coupled, that is, they are combined into pairs[1]. One pole carries the syntactic information while the other pole carries the semantic information. Each unit in a pole (typically[2]) corresponds to an equally named unit in the other pole. For example, the (semantic) unit above may be coupled with the following (syntactic) unit:

```
((unit-1 (syn-cat ((lex-cat Propernoun)))))
```

Again, this is a feature structure that contains one single unit. This unit corresponds to *unit-1* in the semantic feature structure and therefore has been given the same name. The syntactic information carried by this unit consists of the associated slot `syn-cat`. Any slot with this name contains constraints that specify the syntactic category associated with the respective unit. Roughly speaking, slots that are associated with units can be seen as categorization of units, with their specific values to constrain them. These constraints may themselves be slots (or subcategories) that can be constrained by slot fillers. These slots and constraints are further specified in section **??**.

These coupled feature structures allow for a uniform representation of all linguistic knowledge, compatible with current theories of Construction Grammar and beyond.

7.1.5 Handling Hierarchy: the J-Operator

Representing and creating hierarchy in Fluid Construction Grammar is achieved by the *J-operator*. Units marked with the J-operator are ignored during the matching process. When a match is successful, a new unit is introduced and bound to the first argument of the J-operator. The second argument is the parent unit from which the new unit depends (and should therefore already have been bound by the matching process). The third argument specifies the set of units that will be pulled into the newly created unit. The new unit can also contain additional slot specifications, specified in the normal way, and all variable bindings resulting from the match are still present.

7.2 Rule Sets

The rules of an agent are stored in different "rule sets". Each of these rule sets has a "type", which determines

- When the rules are applied. Rule sets can be applied in a specific order, see section [todo: refer to the definition of `fcg-default-production-task` and `fcg-default-interpretation-task`].
- How rule sets are applied, see section 7.4.

---

[1] This information is used by the core engine during the matching and merging processes (see section ??).

[2] TODO: is this a requirement?

- How other operations on rules are performed. Almost all functions that do something with rules can be specialized on the rule set type.

7.2.1 *class* **rule-set**

description

A collection of rules of a same type.

slot **`type`**

```
(type :type symbol :reader rule-set-type :initarg :type)
```

The type of the rule set.

slot **`rules`**

```
(rules :type list :accessor rules :initarg :rules :initform nil)
```

The rules of the rule set. A list of rules.

slot **`left-bins`**

```
(left-bins :type list :accessor left-bins
           :initarg :left-bins :initform nil)
```

A list of "bins" being lists of rules. Each rule that is in the `rules` slot is also in exactly one left bin. That two rules are in the same left bin means that their left poles are equivalent in the sense that they would always match together in production. This information is kept in these bins in order to speed up the application of rule sets.

Normally you will not have to deal with these bins. You don't have to worry about keeping the bins in sync with the `rules` slot, because this is automatically taken care of. See section 7.2.3.

slot **`right-bins`**

```
(right-bins :type list :accessor right-bins
            :initarg :right-bins :initform nil)
```

Similar to slot `left-bins`. Two rules being in the same right bin means that their right poles are equivalent.

copying a rule set

The language game framework uses the generic function `copy-object` (see section [todo: reference]) to copy rule sets. In the implementation of the method for `rule-set`, only the lists of `rules`, `left-bins` and `right-bins` are copied, the rules themselves not.

It is important to remember this when for example changing a rule of a task. The rule sets of a task are copies of the rule sets of the agent (see section [todo: reference]), but the rules themselves directly point to those of the agent. So a good thing here would be to remove the rule from the task and add a modified copy again (which is done automatically by function `replace-rule`, section 7.2.2.8).

7.2.2 Accessing Rules

All the functionality for accessing the rules of a rule set are implemented as generic functions that specialize on a general `object`. Due to this, there are for example methods that remove a rule from a rule set, a rule set collection, a task or an agent (or whatever other object that contains rule sets).

7.2.2.1 *generic function* **add-rule** *object rule*

description	Adds a rule to `object`.
`object`	Anything that contains rule sets. This an be a `rule-set` itself, a `rule-set-collection` (section 7.3), a `fcg-task` (section [todo: reference]), a `fcg-agent` (section [todo:reference]) etc.
`rule`	The rule to add. This can be either an instance of class class `rule` (section **??**) or a list containing 1. the type of the rule (a symbol), 2. the name of the rule (a symbol), 3. the left pole of the rule (a list), 4. the symbol `<-->`, 5. the right pole of the rule (a list).
default implementation	When `object` is a `rule-set`, the rule is directly added to it. This is what happens: • When a rule with the same name already exists in `rule-set`, the previous rule is replaced with `replace-rule` (section 7.2.2.8). • When a rule is “equivalent” to a previous rule, that means that there is already a rule that would be in the same left and right bin, the rule is not added. • The rule is added to the `rules` slot. • The rule is added to the corresponding left and right bins. If there are non yet, new bins are started. If you want to change or extend this behavior, it is recommended to write `:after` or `:around` methods for `add-rule`. For all other types of `object`, the generic function `get-rule-set` (section 7.3.3) is called to get the rule set of `object` and the rule is added to that.

example This two expressions are synonymous:

```
(add-rule *some-task*
  '(lex-stem [box]
    ((?top (meaning (== (prototype ?prototype [box]))))
     ((J ?new-unit ?top)
      (meaning (== (context ?prototype)))))
    <-->
    ((?top (syn-subunits (== ?new-unit)))
     (?new-unit (form (== (stem ?new-unit "box")))))))

(add-rule *some-task* 'lex-stem
          (make-instance
           'rule :name '[box] :type
           :left-pole `((?top
                         (meaning
                          (== (prototype ?prototype [box]))))
                        ((J ?new-unit ?top)
                         (meaning (== (context ?prototype)))))
           :right-pole `((?top
                          (syn-subunits (== ?new-unit)))
                         (?new-unit
                          (form (== (stem ?new-unit "box")))))))
```

The next three expressions are also synonymous, which illustrates the remark at the beginning of section 7.2.2:

```
(add-rule *my-agent* *rule*)
(add-rule (get-rule-set *my-agent* 'con) *rule*)
(add-rule (get-rule-set (get-data agent 'rule-sets)
          *rule*))
```

7.2.2.2 *monitor event* **rule-added** *(rule-set rule-set) (rule rule)*

description	Notified at the end of `add-rule`.
`rule-set`	The rule set that the rule was added to.
`rule`	The added rule.

7.2.2.3 *monitor event* **rule-not-added** *(rule-set rule-set) (rule rule)*

description	In `add-rule`: when a rule was not added because there was already an equivalent rule.
`rule-set`	The rule set.
`rule`	The rule that was not added.

7.2.2.4 *generic function* **get-rule** *object rule-set-type rule-name*

description/ default imlementation	Returns a rule from `object` by its name. Returns `nil` if the rule is not there.
`object`	A `rule-set` or anything that contains rule sets.
`rule-set-type`	The type of the rule set.
`rule-name`	The name of the rule.
example	`(get-rule *some-task* 'lex-stem '[box])`

7.2.2.5 *generic function* **get-rules** *object rule-set-type*

description/ default implementation	Returns the rules of a rule set of `object` as a list.
`object`	A `rule-set` or anything that contains rule sets.
`rule-set-type`	The type of the rule set.
example	`(get-rules *my-agent* 'lex-stem)`

7.2.2.6 *generic function* **remove-rule** *object rule*

description/ default implementation	Removes a rule from a rule set of `object`.
`object`	A `rule-set` or anything that contains rule sets.
`rule`	Either a `rule` instance or a rule name. You will get an error if the rule is not in the rule set.
example	These two lines are synonymous:

```
(remove-rule *some-task* 'lex-stem '[box])
(remove-rule *some-task* 'lex-stem
             (get-rule *some-task* 'lex-stem '[box]))
```

7.2.2.7 *monitor event* **rule-removed** *(rule-set rule-set) (rule rule)*

description	Notified at the end of `remove-rule`.
`rule-set`	The rule set that the rule was removed from.
`rule`	The removed rule.

7.2.2.8 *generic function* **replace-rule** *object previous-rule rule*

description/ default implementation	Added for convenience to replace a rule. First calls `remove-rule` on `previous-rule` and then `add-rule` on `rule`. The names of `previous-rule` and `rule` have to be the same.

`object`	A `rule-set` or anything that contains rule sets.
`previous-rule`	Either a `rule` instance or a rule name.
`rule`	Either an instance of `rule` or a list as in section 7.2.2.1.

7.2.2.9 *monitor event* **rule-replaced** *(rule-set rule-set) (previous-rule rule) (rule rule)*

description	Notified at the end of `replace-rule`. Note that there will be also notifications for `rule-removed` and `rule-added`.
`rule-set`	The rule set.
`previous-rule`	The removed rule.
`rule`	The added rule.

7.2.2.10 *generic function* **print-rules** *object rule-set-type*

explanation/ default implementation	Prints all rules of a rule set of `object`.
`object`	A `rule-set` or anything that contains rule sets.
`rule-set-type`	The type of the rule set.

7.2.2.11 *generic function* **clear-rules** *object rule-set-type*

explanation/ default implementation	Removes all rules from a rule set of `object`.
`object`	A `rule-set` or anything that contains rule sets.
`rule-set-type`	The type of the rule set.

7.2.3 Organizing Rules in Bins

The rules of a rule set are automatically stored in bins (see also section 7.2.1) in order to speed up the application of rule sets. The functions in this section are related to these bins.

As you normally will not get in touch with them, you can easily skip this section.

7.2.3.1 *generic function* **determine-left-bin** *rule-set rule-set-type rule*

description	Determines to which left bin `rule` would belong to. The rule is not added to the bin. Returns the bin (list of rules) or `nil`.
`rule-set`	A rule set.

`rule-set-type`	The type of the rule set. This parameter is not used in the default implementation but this enables you to change or modify the default implementation for specific rule sets.
`rule`	A rule.
default implementation	Finds the bin for that the left pole of the first rule is equivalent to the left pole of `rule`.

7.2.3.2 *generic function* **determine-right-bin** *rule-set rule-set-type rule*

description	The same as `determine-left-bin` (above) for the right bin.

7.2.3.3 *generic function* **find-left-bin** *rule-set rule-set-type rule*

description	Returns the left bin of `rule`. For that, rule has to be in `rule-set`.
`rule-set`	A rule set.
`rule-set-type`	The type of the rule set. For specializing methods.
`rule`	A rule of `rule-set`.
default implementation	Finds the left bin that contains `rule`.

7.2.3.4 *generic function* **find-right-bin** *rule-set rule-set-type rule*

description	The same as `find-left-bin` (above) for the right bin.

7.2.3.5 *generic function* **equivalent-rule-exists?** *rule-set rule-set-type rule*

description	Determines whether there is a rule in `rule-set` with that `rule` would be in the same left and right bin.
`rule-set`	A rule set.
`rule-set-type`	The type of the rule set. For specializing methods.
`rule`	A rule of `rule-set`.
default implementation	Determines the left bin and the right bin for `rule`. Returns `t` when there is any rule that is both in the left and the right bin.

7.3 Collections of Rule Sets

All the rules in different rule sets constitute the linguistic knowledge of an agent. For convenience, the class `rule-set-collection` is defined as a container for different rule sets.

7.3.1 *class* **rule-set-collection**

description	Stores rule sets of different types together.
slot `rule-sets`	`(rule-sets :type hash-table :initform (make-hash-table))` The rule sets.

7.3.2 *generic function* **add-rule-set** *object rule-set*

description/ default implementation	Adds a rule set to `object`. If there is already a rule set of the same type, then this rule set is overwritten.
`object`	A `rule-set-collection` or other objects that contain rule sets, e.g. `fcg-agent` (section [todo: reference]) or `fcg-task` (section [todo: reference]).
`rule-set`	The rule set to add.

7.3.3 *generic function* **get-rule-set** *object rule-set-type*

description/ default implementation	Returns a rule set from `object`.
`object`	A `rule-set-collection`, a `fcg-agent`, a `fcg-task`, etc.
`rule-set-type`	The type of the rule set to return.
example	`(get-rule-set *my-agent* ’morph)` `(get-rule-set *some-task* ’con)`

7.4 Applying Rule Sets

7.4.1 *class* **rule-set-application-spec**

todo:	document

7.4.2 *generic function* **apply-rule-set** *object rule-set-type current-structure direction*

description	Applies a rule set to a coupled feature structure and returns a list of `rule-set-application-spec`s. This is function is main interface to the FCG core both for production and interpretation. You will have to implement this function for each rule set type.
`object`	A `rule-set` or anything that contains rule sets, for example `rule-set-collection` (section 7.3.1), `fcg-agent` (section **??**) or `fcg-task` (section **??**).

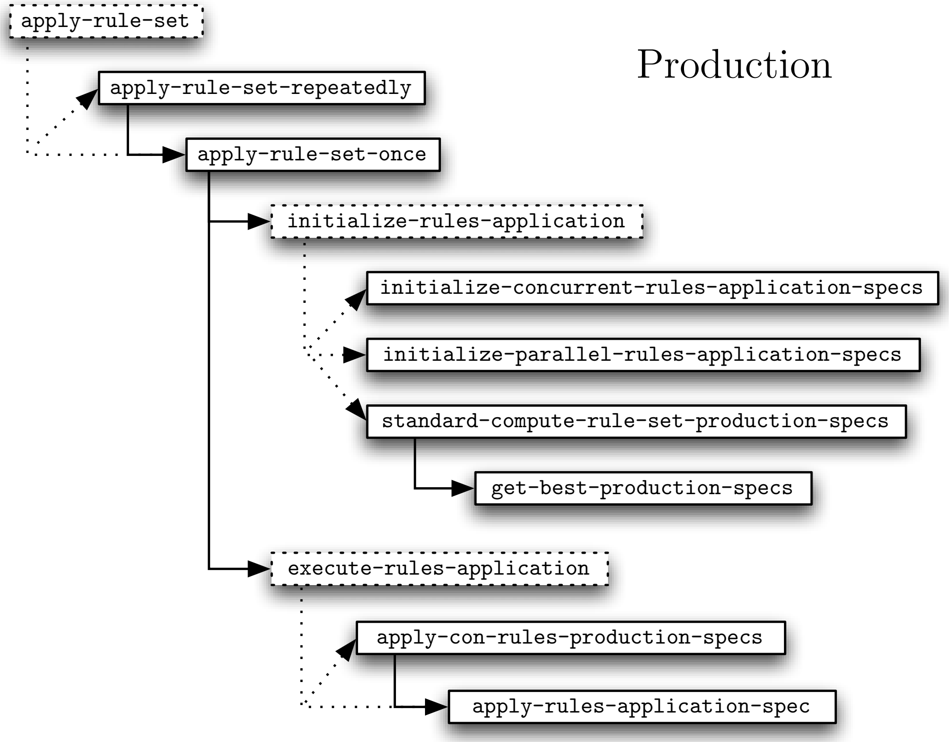

Figure 7.1: Recommended call graph for applying rule sets during production. Functions in a dashed box have to be implemented for each rule set type, functions in a solid box can be changed/ extended.

`rule-set-type`	The type of the rule set, mainly used to specialize methods.
`current-structure`	The coupled feature structure that the rule set is applied to.
`direction`	Whether to produce (`'->`) or to parse (`'<-`).
recommended implementation	Of course you can implement the method for your rule set type as you like, but if you want to follow the recommended scheme from the beginning of this section, you call here either `apply-rule-set-repeatedly` (see below section 7.4.3) or `apply-rule-set-once` (section 7.4.4).
example	todo: examples

7.4.3 *generic function* **apply-rule-set-repeatedly** *todo: parameters*

7.4.4 *generic function* **apply-rule-set-once** *todo: parameters*

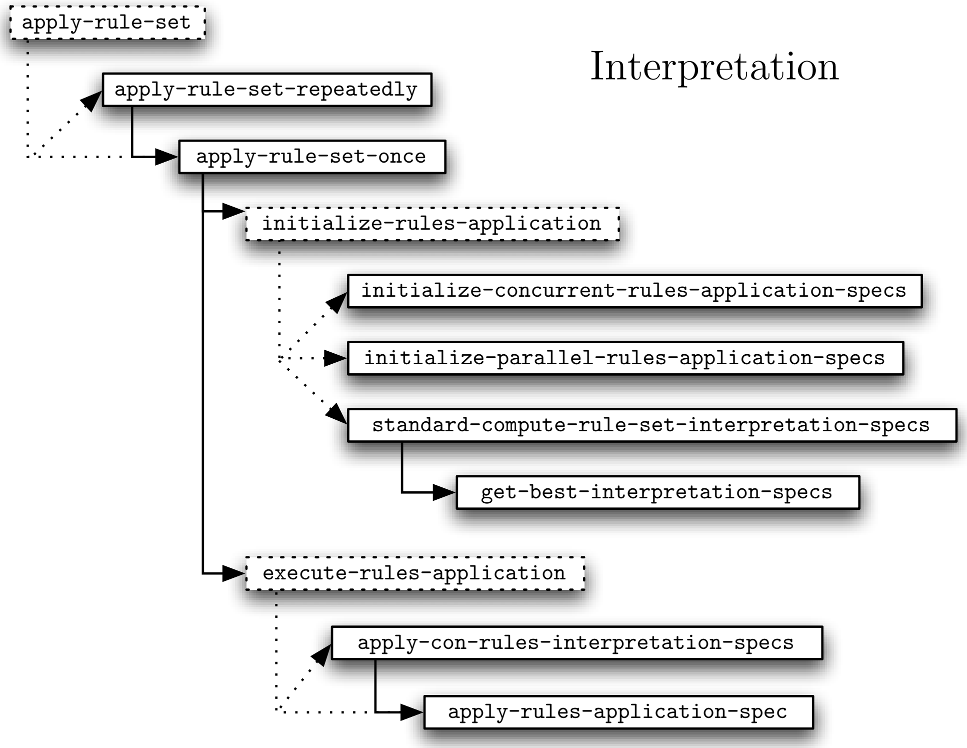

Figure 7.2: Recommended call graph for applying rule sets during interpretation.

8 Test Framework

We supply a home-made, light weight *unit testing framework* that makes use of the powerful conditions system in lisp and the monitoring system from Babel.

In this chapter we will not describe its inner working but only its outward functionality. All of the code can be found in tests/test-framework.lisp.

8.1 Writing tests

Instead of using defun of defmethod when writing unit tests you use deftest. It behaves just like a defun but is surrounded by error-capturing and reporting functionality. In a deftest you can do whatever you would normally do in a defun with the addition of `test-error`, `test-ok` and `test-assert` which you can wrap around other calls.

8.1.1 *macro* **test-error** *expression*

description	test-error should be used for testing that errors you expect to be thrown, are indeed thrown. When wrapping an expression in test-error it will expect an error to be thrown. It will capture any error that is thrown, print a dot and continue. If however no error is thrown, it throws an error itself.

8.1.2 *macro* **test-ok** *expression*

description	When wrapping an expression in test-ok you expect that no error will be thrown. If an error should be thrown it will be captures and it's message will be printed later on. Furthermore an "x" will be printed instead of a dot.

8.1.3 *macro* **test-assert** *expression*

description	When wrapping an expression in test-assert you expect the expression to not throw an error and return not nil. If however nil is returned this will be printed later on. If an error should be thrown this will also be reported.

The nice thing about these three macros is that they give informative feedback when things go wrong and it will continue its processing even if errors are thrown. If all goes well only dots will be printed for every call to one of those macros.

8.2 Example

```
(defun func-that-throws-an-error ()
  (error "foo bar"))
(defun func-that-throws-no-error ()
  t)

(deftest test-1 ()
  ;; these are good tests
  (test-assert (and (equal 2 (+ 1 1))
     (eql (* 2 5) 10)))
  (test-ok (find 1 '(1 2 3)))
  (test-error (func-that-throws-an-error))
  ;; the next ones go wrong, but indeed the code keeps running
  (test-assert (equal 1 2))
  (test-ok (func-that-throws-an-error))
  (test-error (func-that-throws-no-error))
  ;; and again some good tests
  (test-assert 1))

(deftest test-2 ()
  ;; even when an error is thrown outside a (test-... ) call we do not
  ;; crash but we cannot however simply continue.
  (test-ok (+ 1 1))
  (func-that-throws-an-error)
  (test-ok (+ 2 2)))


(defun run-tests ()
  (test-1)
  (test-2))

(run-tests)
```

Calling run-tests will return the following output:

```
TEST-1: ...x
  assertion failed for: (EQUAL 1 2)x
  call: (FUNC-THAT-THROWS-AN-ERROR) generated an error!x
  call: (FUNC-THAT-THROWS-NO-ERROR) DID NOT generate an error but you expected one!.
TEST-2: .x
  TEST-2 threw an unexpected error:
  foo bar
```

Bibliography

J. Bleys. Expressing second order semantics and the emergence of recursion. In A. D. M. Smith, K. Smith, and R. F. i Chancho, editors, *The Evolution of Language: Evolang 7*, pages 34–41. World Scientific, 2008.

W. Croft. Logical and typological arguments for radical construction grammar. In J.-O. Östman and M. Fried, editors, *Construction Grammars: Cognitive Grounding and Theoretical Extensions*, pages 273–314. John Benjamins, Amsterdam, 2005.

A. E. Goldberg. *A Construction Grammar Approach to Argument Structure*. University of Chicago Press, Chicago, 1995.

P. Hopper. Emergent grammar. *Berkely Linguistics Conference (BLS)*, 13:139–157, 1987.

P. Kay and C. J. Fillmore. Grammatical constructions and linguistic generalizations: The what's x doing y? construction. *Language*, 75:1–33, 1999.

R. W. Langacker. A dynamic usage-based model. In M. Barlow and S. Kemmer, editors, *Usage-Based Models of Language*, pages 1–63. Chicago University Press, Chicago, 2000.

M. Loetzsch, R. van Trijp, and L. Steels. Typological and computational investigations of spatial perspective. In I. Wachsmuth and G. Knoblich, editors, *Modeling Communication with Robots and Virtual Humans*, volume 4930 of *Lecture Notes in Computer Science*, pages 125–142. Springer, Berlin, 2008.

S. Russel and P. Norvig. *Artificial Intelligence, A Modern Approach*. Prentice Hall, 1995.

L. Steels. Language games for autonomous robots. *IEEE Intelligent Systems*, pages 16–22, October 2001.

L. Steels. A self-organizing spatial vocabulary. *Artificial Life*, 2(3):319–332, 1995.

L. Steels and F. Kaplan. Situated grounded word semantics. In T. Dean, editor, *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 862–867, Stockholm, Sweden, August 1999. Morgan Kaufmann.

L. Steels and M. Loetzsch. Perspective alignment in spatial language. In K. R. Coventry, T. Tenbrink, and J. A. Bateman, editors, *Spatial Language and Dialogue*. Oxford University Press, 2008. To appear.

L. Steels and P. Wellens. How grammar emerges to dampen combinatorial search in parsing. In P. Vogt, Y. Sugita, E. Tuci, and C. Nehaniv, editors, *Symbol Grounding and Beyond: Proceedings of the Third International Workshop on the Emergence and Evolution of Linguistic Communication, EELC 2006*, volume 4211 of *Lecture Notes in Computer Science*, pages 76–88, Rome, Italy, September 2006. Springer Verlag.

L. Steels, R. Van Trijp, and P. Wellens. Multi-level selection in the emergence of language systematicity. In *Proceedings of the Ninth European Conference on Artificial Life*, volume 4648 of *LNAI*, pages 425–434, Berlin, 2007. Springer-Verlag.

M. Tomasello. *Constructing a Language. A Usage Based Theory of Language Acquisition.* Harvard University Press, 2003.

R. van Trijp. The emergence of semantic roles in fluid construction grammar. In A. D.M. Smith,

K. Smith, and R. Ferrer i Cancho, editors, *Proceedings of the 7th International Conference on the Evolution of Language (EVOLANG 7)*, pages 346–353, Singapore, 2008. World Scientific Publishing.

P. Wellens, M. Loetzsch, and L. Steels. Flexible word meaning in embodied agents. *Connection Science*, 20(2 & 3):173–191, June 2008.

Index