Notice

This paper is the author's draft and has now been published officially as:

Loetzsch, Martin (2012). Tools for Grammar Engineering. In Luc Steels (Ed.), *Computational Issues in Fluid Construction Grammar*, 37–47. Berlin: Springer.

BibTeX:

```
@incollection{loetzsch2012tools,
   Author = {Loetzsch, Martin},
   Title = {Tools for Grammar Engineering},
   Pages = {37--47},
   Editor = {Steels, Luc},
   Booktitle = {Computational Issues in {Fluid Construction Grammar}},
   Publisher = {Springer},
   Series = {Lecture Notes in Computer Science},
   Volume = {7249},
   Address = {Berlin},
   Year = {2012}}
```

Tools for Grammar Engineering

Martin Loetzsch

Artificial Intelligence Laboratory, Vrije Universiteit Brussel, Belgium

Abstract. Developing a FCG grammar can easily become very complex when larger sets of interdependent constructions are involved and when consequently tracking down mistakes in single constructions or analyzing the overall behavior of the system becomes difficult. FCG supports users in this task by providing interfaces for accessing most of its internal representations together with powerful debugging and visualization tools. This paper will outline some of these interfaces and explain its visualization components in detail.

1 Introduction

Fluid Construction Grammar is a formalism for defining grammars along the lines advocated by construction grammarians and a theory and implementation on how sentences are parsed and produced using construction grammar [4]. Unlike many other implementations of grammar formalisms, FCG does not come with a standalone program that is started once and then provides a (possibly graphical) user interface for editing and testing grammars – FCG is rather a software framework that users directly interact with. FCG is written in the programming language Common Lisp [3], which is commonly used in artificial intelligence and computational linguistics. The choice of Lisp makes it very easy to adapt FCG to a variety of usage scenarios because Lisp allows for powerful yet simple abstractions that flexibly wrap core mechanisms for specific uses. Lisp makes it also easy to write *macros* that allow the definition of constructions by specifying only their relevant features (and thus avoid writing redundant code).

In addition to the increased flexibility of use, the choice of Lisp as a programming language means that the traditional time consuming "edit - compile - test cycle" (i.e. changing code, then compiling the program and then testing it) does not exist. Instead, users load the FCG framework into a Lisp programming environment and then on top of that develop and test their grammar or multi-agent experiment. During that, the "Lisp image" (i.e. the "program") is constantly running and changes (e.g. adding constructions or changing functions) have immediate effect on the state of the system.

A minimal example of a typical interaction with FCG is shown in Figure 1 (see [5] for a first introduction to FCG). The GNU Emacs text editor (shown on the right) serves here as a Lisp environment, but any other Lisp IDE can be used. The Emacs window contains a Lisp file with a variety of Lisp expressions that each are manually evaluated by the user to change the state of the Lisp image (but such a file can also be loaded and evaluated automatically as

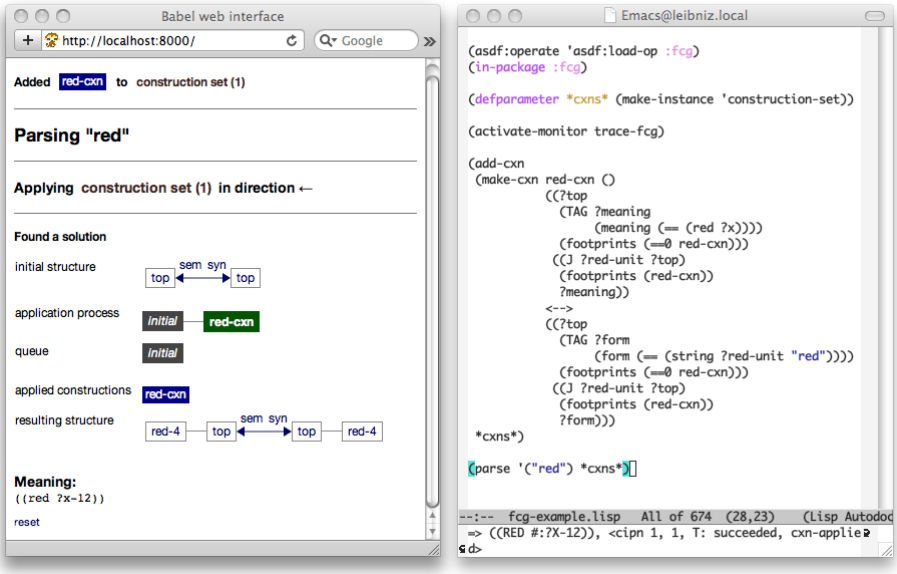

Fig. 1. Interacting with FCG. On the right, a Lisp development environment (here GNU Emacs) containing a minimal example of using FCG is shown. The result of evaluating some of the expressions in the example (such as adding a construction to a set and parsing an utterance) is then visualized in real-time through a standard web browser (here Safari, on the left).

a part of a larger application). The first two expressions `(asdf:operate ..)` and `(in-package ..)` load the FCG framework into the current Lisp image so that from then on it can be used by the user. Then an empty construction set is created `(defparameter ..)` and a monitor for visualizations is activated. The fifth expression `(add-cxn ...)` adds a new construction to the previously created construction set and the notification "Added red-cxn to construction-set (1)" appears in the web browser window. Finally, the utterance "red" is parsed and some graphical output of FCGs processing appears in the web browser. Changing the previously defined construction or adding new constructions does not require to repeat all the steps above because the Lisp image is still running and thus developing a grammar is a continuous process of adding or replacing constructions and then trying them out.

The learning curve for new users of FCG is quite high because they have to learn how to work with a Lisp environment. However, there are detailed instructions for getting FCG running on all major platforms / Lisp implementations and once this initial hurdle is taken, interacting with FCG becomes an inspiring experience. Part of this experience is an extensive visualization and debugging component, which we will introduce in Section 2 and explain in a bit more in Section 3.

2 Understanding FCG processing

Producing or parsing utterances in FCG involves complex processes that depend on the interplay of the involved constructions as well as often unpredictable external systems for discourse, semantics and learning. Consequently, understanding the behavior of such processes may become quite a challenge. For example it can be very difficult to track down which construction needs to be changed when a production or parsing process fails, or to analyze which construction is responsible for an undesired utterance or meaning. Even when the behaviour is as expected it might be far from trivial to understand or show why this is the case, or for example how a grammar could be further refined. Furthermore, when FCG is applied in multi-agent learning scenarios where constructions are continuously created, adapted or removed by agents or where systems for embodied semantics provide open-ended streams of conceptual structures, the behavior of FCG grammars can not be tested in isolation because it tightly interacts with all these other subsystems. These sorts of difficulties in understanding the underlying dynamics of complex software systems are a common problem in many areas of computer science such as robotics, artificial intelligence and distributed systems.

The probably most common approach to tackle this problem is *tracing*, i.e. by printing information about what is going on to to the Lisp listener (either by directly adding print statements to the code or by using built-in trace facilities or other custom mechanisms). The advantages of this technique are that (1) it is very easy to do and (2) it allows to monitor program execution at almost any level of detail (i.e. from only the result of applying a construction set to near-complete

information including all intermediate processing steps). When the level of detail is high however, the output rapidly becomes unmanageable: complete traces of FCG search processes can reach hundreds of text pages. Furthermore, plain text in a Lisp listener is not easy to read and can only be presented linearly which makes it impractical for getting an overview of, or a 'feeling' for, particular dynamics of FCG.

A second method is to retrospectively analyse a search process by *inspecting* Lisp objects – either using inspector tools of Lisp environments or by directly calling chains of accessor functions on an object. As we will explain further below, FCG allows users to access detailed data representations of most of its internal processing steps so that the complete chain of operations can be reconstructed. Inspection has the advantage that it does not require to change or write any code but on the other hand does not provide any overview of a process whatsoever. Furthermore, accessing for example a node deep in a search process requires a high number of manual steps and often it is unclear which one of the many objects to inspect, which prevents inspection to be useful in many cases.

Third, many Lisp implementations and development environments provide mechanisms for manually *stepping* through code either by using the built in step facilities or by invoking the stepper directly from the code. This technique can be very helpful for finding logical mistakes in small pieces of code, but in order to make use of stepping one has to know which part of the code to look at. Stepping through a large construction set application just for exploring its dynamics would take hours and is thus impracticable.

A last technique is to create *visualisations*: they are great to get an intuition of an algorithm's dynamics because our mind understands graphical representations much better than text output, which is why in areas such as for example robotics it is exceptional to even write programs without having graphical means to verify that the system behaves as expected (ideally even for each intermediate step). Visualisations are costly, because it takes time to implement them, but they often pay off in the long run when they become an invaluable eye into the system's internals. A clear disadvantage is that visual representations only allow for a low to medium level of detail: complex data structures have to be transformed into two-dimensional (or sometimes 3D) representations, involving constraints such as available window size and a required visual clarity so that the representation remains readable. Furthermore, despite many recent developments in cross-platform libraries for graphical user interfaces, there is none that easily works in all major Common Lisp implementations and there is a lot of overhead in dealing with windows, menus and other user interface elements, event handling and the interaction with the actual code to monitor. Finally, FCG processing relies on many textual (i.e. semantic structures) and hierarchical (e.g. feature structures, search processes) data structures, which in turn are hard to visualise. Although virtually every graphic library has means to display text on the screen, the responsibility for arranging text blocks (estimating widths an heights of text areas depending on available space, avoiding overlap, re-flowing

multi-line text) is usually in the hand of the programmer.

The implementation of the FCG formalism comes with an extensive user interface (called *web interface*) that combines many of the advantages of the previously introduced techniques, while at the same time removing most of their respective disadvantages. The main idea (we will go into some of the details in the next section) is to use a normal web browser as a terminal for tracing internals of FCG. That is, information is "printed" in real time to the web browser window instead of to the Lisp listener. This already has the advantage that different colors, text styles, backgrounds and many other graphical means can be used to make representations more readable than if there were printed as plain text. Second, recent advances in HTML rendering engines have made it possible to create tree-like representations (for feature structures, search trees, etc.) that are automatically laid out by the client browsers so the available screen space is used as economically as possible. And third, scripting mechanisms of contemporary web browsers allow to create expandable and collapsible elements so that details of a structure or a process can be initially hidden but then be recursively revealed if the user clicks on them, thus enabling to display the most important information of a structure or process in a single browser window, while still providing access to the smallest details. Forth and finally, current web technologies enable to interact with FCG grammars through the browser window (e.g. applying constructions to previously selected transient feature structures with a mouse click).

An example for such interactive visualizations is shown in Figure 2. The utterance "block" is parsed and as a side effect, a condensed visualization of the involved processes and representations is sent to the web browser. This high-level summary shows the utterance, the applied construction set, the initial coupled feature structure, a visualization of application search tree, the applied constructions, the final coupled feature structure and finally the resulting semantic structure are shown. This information is already enough to get a good overview of what happened, that is, whether the processing succeeded, which constructions were applied in which order and what the resulting utterance or semantic structure is. Showing only this little information has the advantage that even the processing of larger utterances or meanings can still be visualized in a single browser window.

This information is already enough to verify whether the grammar behaves as expected. However, when developing a set of constructions for a particular linguistic example, this is usually not the case and finding out which constructions need to be changed to yield the desired result is far from trivial. FCG helps with this by making virtually all of its internal representations and intermediate processing results accessible through graphical representations that reveal a more detailed (and bigger in terms of screen space) version when the user clicks on them. For example, each node in a application search initially only shows only the name of the applied construction (which results in a compact representation of the search tree), but when expanded, near-complete information about

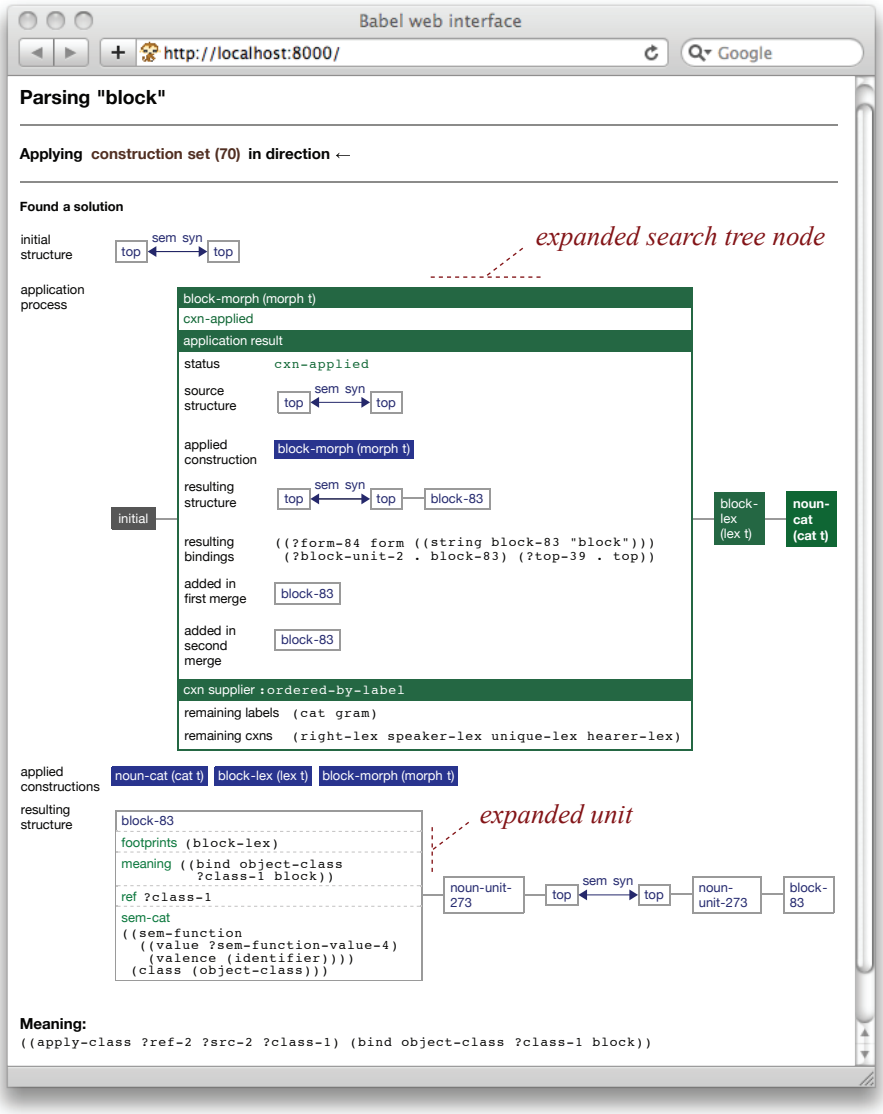

Fig. 2. Example output produced by the web interface. The search tree node (`block-morph`) and unit (`block-83`) of the resulting structure have been manually expanded.

the construction application, goal tests and other internals of the application process are shown. Concretely, as shown in the center of Figure 2, the expanded versions of search tree nodes contain some search status information, the transient structure to which the construction was applied, the applied construction, the resulting transient structure, bindings that show what things the variables in the construction matched with and information about what was added to which unit in the first and second merge phase. Many of these representations such as constructions or transient structures can in turn be further expanded, as for example shown at the bottom of Figure 2, where the unit `block-83` of the final coupled feature structure is expanded to show all of its features. Sometimes, when a goal test requires running another construction set application, the whole search tree of that process will be also shown within that particular node, leading to quite deeply nested visual representations. This exploratory aspect of the interaction with FCG is best experienced firsthand and readers are invited to try out the examples at `www.fcg-net.org`.

In addition to allowing users inspect most processing details of FCG, some representations can also be manipulated through the web interface. When the mouse is moved over the graphical representation of a construction, a coupled feature structure, or a search node, a small menu appears that offers a set of operations on the structure:

For search nodes, these operations include rendering the transient structure of the node into an utterance or extracting its meaning, saving the actual node or its transient structure to a global variable so that it can be accessed from within Lisp, and restarting the search process below the node. Furthermore, the pop-up menu of constructions contains operations for saving and printing the construction as well as for applying the construction to a previously saved transient structure in parsing or production. Finally, coupled feature structures also can be saved, printed, rendered, etc. – additionally there are operations for applying all constructions of the construction inventory to the structure in order to see which constructions apply and which not.

All these mechanisms help FCG users to quickly find and fix problems in their grammars. For example, in order to find out why a construction that should have been applied did not apply, the expandable graphical representations of construction application search trees make it easy to find the last "correct" transient structure, which can be saved through its menu to a global variable and then can be compared to the construction in question side by side. The changes to the construction are made in the Lisp editor, but as soon as the construction is added to a construction inventory, a graphical representation of the changed construction will appear in the web browser, where it can be applied to the saved transient structure through its popup menu. This isolated application test is continued until the construction behaves as expected and can be tried out again within a complete production or parsing process.

3 The web interface and monitoring system

FCG shares its visualization and monitoring mechanisms with other systems for discourse, semantics and learning under the umbrella of the Babel2 framework [2, 6]. The web interface is described in detail in [1] and the monitoring system in [2, Chapter 3], but we will nevertheless outline some basic design choices and components in this section.

The web interface is based on the HUNCHENTOOT web server that runs in the same Lisp image as FCG/ Babel2 and is loaded automatically with FCG. Web browsers connect to the web interface by loading an empty client page and then content can be *pushed* to that page from within Lisp through an AJAX-based client-side event loop that frequently polls the web server for new elements to display. Creating HTML elements in Lisp is a straightforward transformation of s-expressions such as `((p :style "color:red") "hi!")` into `<p style="color:red">hi!</p>"` and with these basic mechanisms it is already possible to create program traces that are much more readable and informative than plain text because different colors, a multitude of font styles, sizes, backgrounds or borders can be used.

Contemporary HTML rendering engines are extremely good in distributing the available browser window width among recursively nested child elements, and in re-flowing the layout when the window size changes or when more data is added to the page). FCGs web interface heavily exploits these abilities by providing general mechanisms for drawing tree-like structures (such as search trees, feature structures, constructions) in HTML. Tree nodes and horizontal and vertical lines connecting them are created as recursively nested tables and the web browser then takes care of fitting the tree in the available space and of adjusting the width of the nodes accordingly. Furthermore, as many of FCGs representations are Lisp s-expressions (e.g. semantic structures, feature values of coupled feature structures), and experienced Lisp users strongly rely on proper indentation of parentheses, the web interface can create HTML representations of s-expressions that automatically adopt their width to the available space, while remaining properly indented. As shown in Figure 3, this flexible layout of feature structures (and other tree-like representations) results in a very efficient use of (precious) browser window space.

A second key feature of the web interface is the way in which complex data and control structures can be displayed: the level of detail of what is visualised is not restricted by the size of the web browser window. The trick is to create simplified visual representations of data that *expand* into a detailed version when the user clicks on them and that *collapse* to their original state when clicked a second time. As for example explained above in Section 2, for a production or parsing process the user initially is presented with the main results of the application process and the search tree and then can get to almost any intermediate representation by clicking through the HTML elements in the client page. Technically, the web interface internally stores for each of expandable elements both a collapsed and expanded version (actually anonymous functions are stored to avoid computing HTML code that never gets requested). The collapsed version

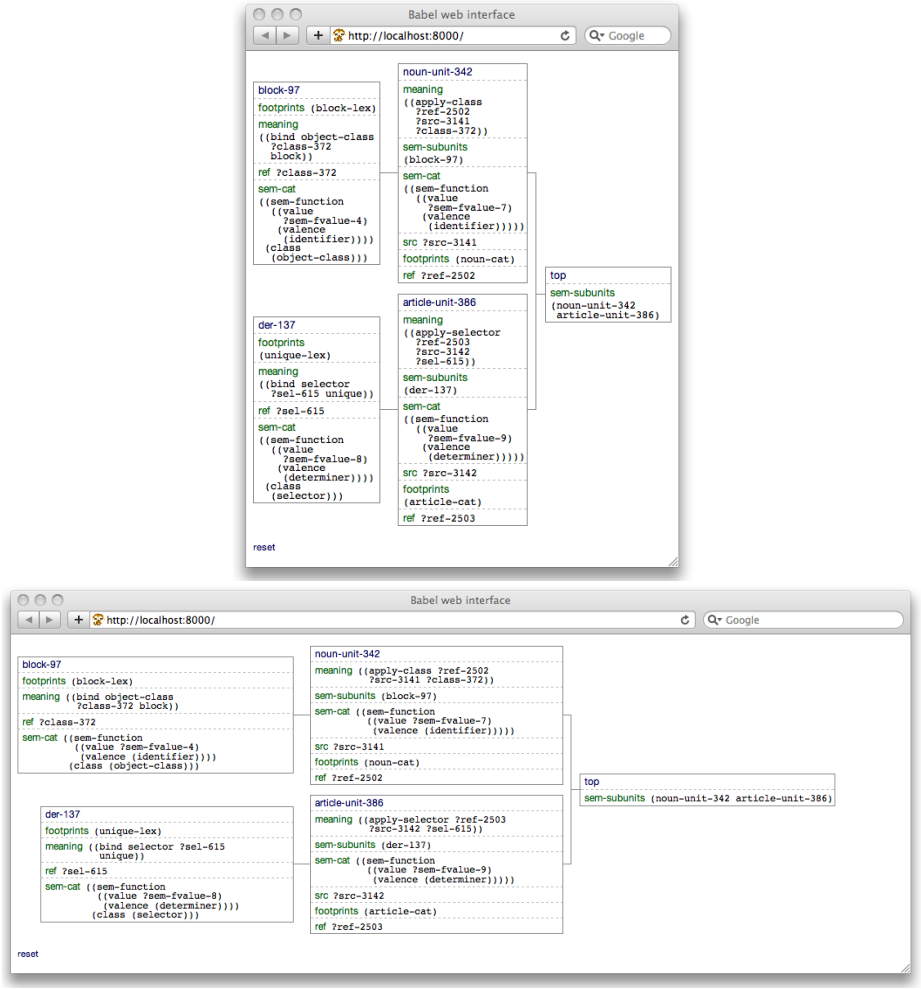

Fig. 3. Example for a resizing HTML representation of a feature structure. The same semantic pole of a transient structure is automatically rendered by the Safari web browser to fit into a horizontally constrained space (top) or a more wide browser window (bottom).

is initially sent to the client page, and when the user clicks on the element the expanded version is requested from the Lisp side using Ajax communication.

The use of lexical closures makes the web interface very fast and responsive, but users may not always want the visualizations to be computed (or switch between alternative visualizations). The *monitoring* system of FCG/ Babel2 solves this issue (for details see chapter 3 of [2]). The main idea is to not clutter the source code of FCGs implementation with code that produces debug traces, but rather to have a event/ notification/ event handler system that separates actual code from debugging and visualization mechanisms. For example, after each application of a construction set, an event is triggered that notifies a list of *monitors* of the finished application. Those monitors that are *active* can then *handle* this event, for example by adding visualizations to the web interface. Consequently, FCG users can switch different visualizations on and off by activating/ deactivating monitors.

Finally, advanced users of FCG and Lisp can easily create their own visualizations and monitors for their specific experiments. The web interface provides mechanisms for defining client side Javascript and CSS code fragments, for replacing and appending the content of existing HTML elements, and so on. When the basic graphical capabilities of HTML are not sufficient for certain visualisation purposes, SVG graphics or Flash animations can of course be also sent to the client page.

4 Conclusions

The tools for grammar engineering introduced in this chapter have changed the way users of FCG interact with the system, not only making the experience richer, but also more trouble-free. Meaningful processing traces allow FCG developers to debug and make sense of the formalism itself and they help FCG users to see the effects of their linguistic rules by having access to all details of the parsing process without losing a general overview. Some dynamics that were previously mystifying and impenetrable have cleared up and as a welcome side-effect newcomers to FCG grasp the material at a considerably faster speed because of the way they can interact with and gradually investigate the dynamics. Furthermore, having detailed visualizations is very helpful explaining the working of FCG to other audiences – in fact most of the graphics in this book are visualizations created by the web interface.

Using a web server and web browsers as the main technology for designing a user interface might seem as an unusual choice and indeed only a few years ago it would not have been feasible to implement a system such as described here. Although our present-day web standards are much older, they were only poorly supported making it strenuous for web developers to ensure the interoperability of their web sites in different browsers. Fortunately, things have improved. Today valid XHTML+CSS code is properly interpreted by a variety of browsers and AJAX has become an ubiquitous technology that just works. Particularly impressive is the way contemporary HTML rendering engines are able to lay-

out (and re-flow) heavily nested HTML constructs with incredible speed and perfectly looking in almost all cases. Furthermore, not needing to rely on other external libraries for user interfaces than (readily available) web browsers makes the visualization component of FCG/ Babel2 very lightweight and easily employable on a wide variety of Lisp implementations and platforms.

Acknowledgements

The research in this paper was conducted at the Artificial Intelligence Laboratory at the Vrije Universiteit Brussel (VUB AI lab) and is partly funded through the EU FP7 ALEAR project.

Bibliography

[1] Loetzsch, M., Bleys, J., Wellens, P.: Understanding the dynamics of complex lisp programs. In: Proceedings of the 2nd European Lisp Symposium. pp. 59–69. Milano, Italy (May 2009)
[2] Loetzsch, M., Wellens, P., De Beule, J., Bleys, J., van Trijp, R.: The Babel2 manual. Tech. Rep. AI-Memo 01-08, AI-Lab VUB, Brussels, Belgium (2008)
[3] Steele, G.L.: Common LISP: The Language. Digital Press, Bedford, MA, 2. edn. (1990)
[4] Steels, L. (ed.): Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam (2011)
[5] Steels, L.: A first encounter with Fluid Construction Grammar. In: Steels, L. (ed.) Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam (2011)
[6] Steels, L., Loetzsch, M.: Babel: a tool for running experiments on the evolution of language. In: Nolfi, S., Mirolli, M. (eds.) Evolution of Communication and Language in Embodied Agents, pp. 307–313. Springer-Verlag, Berlin Heidelberg (2010)