PyFCG: Fluid Construction Grammar in Python

Paul Van Eecke*
Artificial Intelligence Laboratory
Vrije Universiteit Brussel, Belgium
paul@ai.vub.ac.be

Katrien Beuls*
Faculté d'informatique
Université de Namur, Belgium
katrien.beuls@unamur.be

Abstract

We present PyFCG, an open source software library that ports Fluid Construction Grammar (FCG) to the Python programming language. PyFCG enables its users to seamlessly integrate FCG functionality into Python programs, and to use FCG in combination with other libraries within Python's rich ecosystem. Apart from a general description of the library, this paper provides three walkthrough tutorials that demonstrate example usage of PyFCG in typical use cases of FCG: (i) formalising and testing construction grammar analyses, (ii) learning usage-based construction grammars from corpora, and (iii) implementing agent-based experiments on emergent communication.

1 Fluid Construction Grammar

Fluid Construction Grammar (FCG – Steels, 2004; van Trijp et al., 2022; Beuls and Van Eecke, 2023) is a computational construction grammar framework that provides a collection of high-level building blocks for representing, processing and learning fully-operational construction grammars. The FCG framework is conceived as an open instrument that is not tied to a particular construction grammar theory, but that strives for compatibility with any linguistic theory that adheres to the most fundamental tenets underlying constructionist approaches to language (see e.g. Fillmore, 1988; Croft, 2001; Goldberg, 2003). As such, it subscribes to the view (i) that language users dynamically build up their own linguistic systems as they communicate with other members of their community, (ii) that these linguistic systems can be captured as a network of form-meaning mappings called constructions, and (iii) that these constructions can pair forms and meanings of arbitrary complexity and degree of abstraction, thereby facilitating a uniform handling of both compositional and non-compositional linguistic phenomena.

FCG is primarily being used as the language representation, processing and learning component in agent-based models of linguistic communication. Such models simulate the emergence, evolution and acquisition of human languages in populations of artificial agents that take part in situated communicative interactions modelled after those that human language users continuously engage in (e.g. van Trijp, 2016; Beuls and Van Eecke, 2024). Other common uses of FCG include the formalisation and computational operationalisation of construction grammar analyses (e.g. Gerasymova, 2012; Micelli, 2012), and the corroboration of construction grammar theories with empirical data (e.g. Moerman et al., 2024).

FCG is being developed as an open-source community project, which brings together the construction grammar and computational linguistics communities. While strong ties between both communities already existed when the field of construction grammar was founded in the 1980s, recent initiatives such as UCxn (Weissweiler et al., 2024) and the Construction Grammars and NLP (CxGs+NLP) workshop series (Bonial and Tayyar Madabushi, 2023), along with an increasing volume of work on constructions in Large Language Models (see e.g. Tayyar Madabushi et al., 2020; Tseng et al., 2022; Weissweiler et al., 2022; Bonial and Tayyar Madabushi, 2024; Xu et al., 2024), bear witness to a growing interest in research at the intersection of both fields.

2 FCG in Python, Really?

Readers who have regularly used FCG might argue that there already exists a stable and mature, efficient, cross-platform and open source implementation of FCG, with an active and dedicated, albeit

*Both authors contributed equally. The authors declare that this paper was conceived and written without the assistance of generative writing aids.

small, developer community[1]. Indeed, the reference FCG implementation is written in Common Lisp, a dynamic and extensible programming language that, admittedly, excellently fits the project's requirements, including multi-paradigm, high-level and multi-threaded programming, fast prototyping, and highly efficient symbol processing. So why would anyone spend time and effort on a Python port?

The reality is that Python has become the most popular programming language in the world[2] and that, more consequentially, it has also become the dominant language in programming education, as well as today's de facto standard in both linguistics and natural language processing research. Practically speaking, this entails that the success of a software library targeted at the broader computational linguistics community depends before all other things on its compatibility with the Python ecosystem. While this might sound utterly unreasonable, it is not entirely so. For one thing, computer programs typically integrate a variety of external libraries, and, for better or for worse, the largest selection of libraries tends to be developed for the most popular programming languages. For another, the investment involved in learning to use a new programming language and environment is a very real obstacle for potential users in today's time-pressed society.

The development and release of PyFCG follows a trend set by many other libraries that are commonly used in the computational linguistics community. For example, the Stanford CoreNLP Java library (Manning et al., 2014) is now accessible from Python through the Stanza library (Qi et al., 2020). Likewise, the PRAAT system for phonetic analysis (Boersma and Weenink, 2025), written in C and C++, is now widely used in Python programs via the Parselmouth library (Jadoul et al., 2018). The Torch library for tensor computation, written in C and Lua, is now even primarily being developed for Python as part of the PyTorch project (Paszke et al., 2019).

3 FCG in Python, Finally!

Readers who have not yet used FCG, perhaps for the reasons mentioned above, might wonder why it has taken so long for FCG to make its way into the Python ecosystem. It could be ascribed to a lack of actual problems with the existing reference implementation, to the sheer size, scope and complexity of its codebase, to the difficulty of funding scientific software development, or perhaps most likely, to a combination of all these factors. But fret no more:

```
$ pip install pyfcg
```

Once pip-installed, PyFCG can readily be imported as a module into Python programs. It is customary to define `fcg` as an alias for the PyFCG module, so that all functionality is available within the `fcg` namespace. We initialise PyFCG by calling its `init()` function, which loads (or downloads if necessary) all external dependencies. PyFCG's documentation is available on the *Read the Docs* platform[3] and interactive tutorial notebooks supporting this paper can be downloaded from the FCG community website[4].

```
>>> import pyfcg as fcg
>>> fcg.init()
```

On the highest level, PyFCG defines three classes that are of interest to the user: `Agent`, `Grammar` and `Construction`. The idea is that an agent (of type `Agent`) has a grammar (of type `Grammar`), which in turn holds constructions (of type `Construction`). The `Agent` class is the main entry point for the user. Upon the creation of a new agent, it is automatically initialised with an empty grammar, i.e. a grammar that holds zero constructions.

```
>>> demo_agent = fcg.Agent()
>>> demo_agent.grammar.size()
0
```

It was an explicit design choice to tie grammars to agents, emphasising FCG's view that a grammar always represents the linguistic knowledge of an individual language user. Grammars are in principle never shared between agents and cannot exist outside an agent. Instances of the `Grammar` class should therefore only be created implicitly via the `Agent` class.

4 PyFCG at Work

We present three walkthrough tutorials that showcase how PyFCG can be integrated in typical use cases of FCG: grammar formalisation and testing

[1] See https://gitlab.ai.vub.ac.be/ehai/babel for the project's code repository.

[2] https://www.tiobe.com/tiobe-index/

[3] https://pyfcg.readthedocs.io

[4] https://fcg-net.org/pyfcg

(4.1), learning grammars from semantically annotated corpora (4.2), and modelling emergent communication (4.3). Each tutorial is accompanied by an interactive notebook, which can be downloaded from the FCG community website[4].

4.1 Grammar formalisation and testing

A common use of FCG revolves around the formalisation and computational operationalisation of construction grammar theories and analyses. Not only can computational operationalisations help validate their preciseness and internal consistency, they also facilitate the comparison, exchange and integration of insights from different researchers (van Trijp et al., 2022). This tutorial exemplifies how PyFCG can be used to equip an agent with a designed grammar, how new constructions can be added to or removed from the agent's grammar on the fly, how the agent can use its grammar to comprehend and formulate utterances, and how all these processes can be visually inspected through FCG's graphical web interface. For more information on aspects of the syntax and semantics of FCG that are not particular to the PyFCG module, we refer the interested reader to Van Eecke (2018, Chapter 3)[5].

After importing and initialising PyFCG, we create a new agent named Sue. Sue, as an instance of the `fcg.Agent` class, is automatically initialised with an empty grammar. Sue is also assigned a unique identifier:

```
>>> sue = fcg.Agent(name='Sue')
>>> sue
<Agent: Sue (id: sue-1) ~ 0 cxns>
```

Sue can read in a predefined grammar, specified in the Open FCG Exchange Format (OFEF). In this case, we use PyFCG's `load_resource` function to download a human-designed demo grammar fragment that specifies six constructions for processing the English resultative sentence "*Firefighters cut the child free*."[6]. This grammar fragment uses the Abstract Meaning Representation format (AMR; Banarescu et al., 2013) to represent constructional meaning[7]. We instruct Sue to load the grammar specified in the file by calling their `load_grammar_from_file` method and then list the names of the constructions that were loaded.

[5]Or alternatively to https://emergent-languages.org/wiki/docs/recipes/fcg/syntax-and-semantics.

[6]Example after Hoffmann (2018).

[7]AMR meaning representation kindly provided by Claire Bonial (p.c. 31/03/2023).

```
>>> f = fcg.load_resource('demo-
    resultative.json')
>>> sue.load_grammar_from_file(f)
>>> sue
<Agent: Sue (id: sue-1) ~ 6 cxns>
>>> list(sue.grammar.cxns.keys())
['firefighters-cxn', 'child-cxn',
 'cut-cxn', 'free-cxn', 'np-cxn',
 'resultative-cxn']
```

We now ask Sue to comprehend the utterance "*Firefighters cut the child free*.". In FCG, comprehension and formulation respectively refer to the processes of mapping utterances to their meaning representation and vice versa. In order to be able to visually inspect Sue's comprehension process, we first start up FCG's graphical web interface and activate the standard `trace-fcg` monitor. After having comprehended the utterance, we visualise the resulting AMR meaning representation in the more human-readable Penman format. We can see that Sue understood that a cutting action in the sense of '*slice, injure*' (denoted by PropBank's `cut-01` roleset; Palmer et al., 2005) was performed by an '*intentional cutter*' (`arg0` in `cut-01`), more in particular a person who habitually '*fights*' (`fight-01`) fire (`arg1` in `fight-01`), and that the cutting action itself led to (`arg0-of` in `cause-01`) the '*unconstrained, unrestricted*' state (`free-04`) of a child (`arg1` in `free-04`). A screen capture of the web interface after comprehending the utterance is shown in Figure 1.

```
>>> fcg.start_web_interface()
>>> fcg.activate_monitors(['trace-fcg'])
>>> amr = sue.comprehend("Firefighters
    cut the child free.")
>>> fcg.predicate_network_to_penman(amr)
'(c / cut-01
      :arg0 (p / person
           :arg0-of (f / fight-01
                :arg1 (f2 / fire)))
      :arg0-of (c2 / cause-01
           :arg1 (f3 / free-04
                :arg1 (c3 / child))))'
```

Let us now add a new construction to Sue's grammar: the DOG-CXN that in essence pairs the form "*dog*" with its AMR meaning of instantiating the dog concept. Along with its name, we specify its contributing and conditional poles, and load it into Sue. We also add a categorial link between the category proper to the DOG-CXN and the category of the noun slot in the NP-CXN, following the design choices made in the demo grammar fragment and very much in the spirit of Radical Construction Grammar (Croft, 2001).

```
>>> dog_cxn = fcg.Construction(
      name= 'dog-cxn',
```

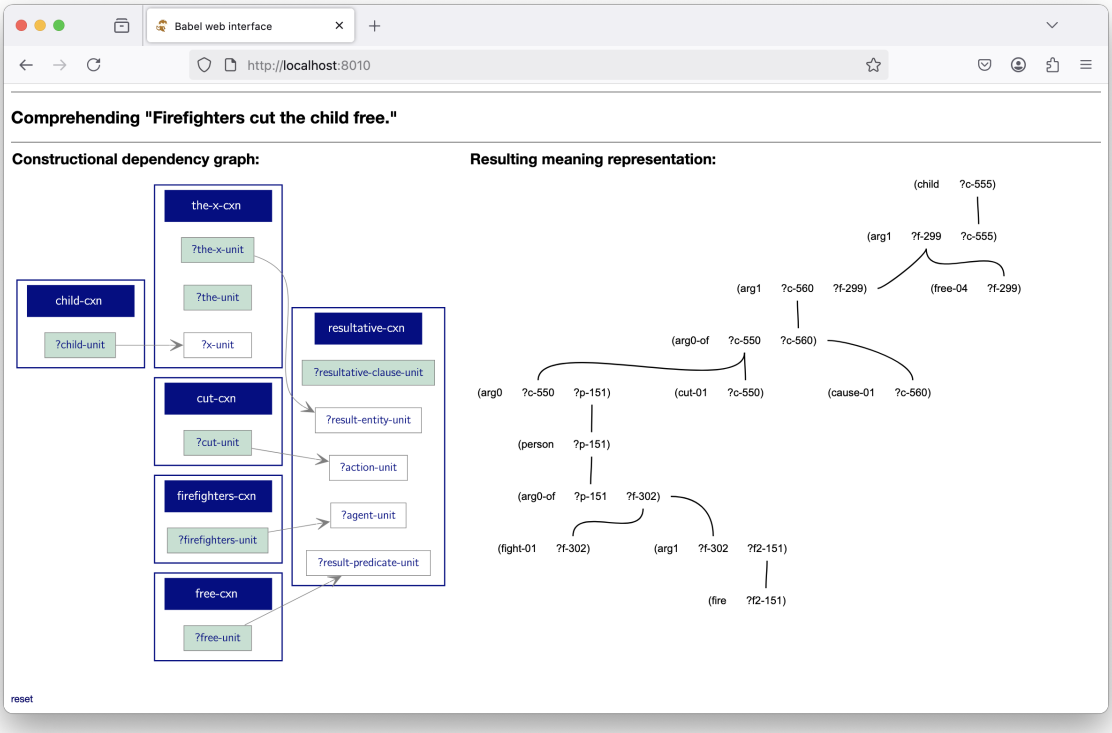

Figure 1: Screen capture of FCG's web interface after calling `sue.comprehend("Firefighters cut the child free.")` with the `trace-fcg` monitor activated.

```
      contributing_pole=
        [('?dog-unit',
          {'referent': '?d',
           'category': 'dog-cxn',
           'boundaries':
                ('?left', '?right')})],
      conditional_pole=
        [('?dog-unit',
          {'#meaning': [('dog', '?d')]},
          {'#form':
             [('sequence', '"dog"',
               '?left', '?right')]})])
>>> sue.add_cxn(dog_cxn)
>>> sue.add_category('dog-cxn')
>>> sue.add_link('dog-cxn','np-cxn-n')
>>> sue
<Agent: Sue (id: sue-1) ~ 7 cxns>
```

We now ask Sue to use their grammar to `formulate` an utterance that expresses that a cutting action in the sense of '*slice, injure*' was performed by an '*intentional cutter*', more in particular a person who habitually '*fights*' fire, and that the cutting action itself led to the '*unconstrained, unrestricted*' state of a dog:

```
>>> amr = '(c / cut-01
             :arg0 (p / person
                 :arg0-of (f / fight-01
                     :arg1 (f2 / fire)))
             :arg0-of (c2 / cause-01
                 :arg1 (f3 / free-04
                     :arg1 (d / dog))))'
>>> amr = fcg.
    penman_to_predicate_network(amr)
>>> sue.formulate(amr)
'Firefighters cut the dog free.' ,
```

We can see that Sue produces the utterance "*Firefighters cut the dog free.*". Again, the construction application process can be traced in detail in the web interface.

4.2 Learning grammars from corpora

A second use case of FCG concerns the learning of construction grammars from corpora of language use. We take the example of `fcg-propbank`, an existing FCG subsystem for learning construction grammars from PropBank-annotated corpora. We demonstrate how a pretrained grammar comprising tens of thousands of constructions can be loaded into an agent and used to extract semantic frames from open-domain text, how a new grammar can be learnt from annotated data, and how large grammars can be saved in an efficiently loadable binary format.

As always, we start by creating an agent. In this case, the agent is an instance of the `fcg.PropBankAgent` class, a subclass of the `fcg.Agent` class provided by the `fcg-propbank`

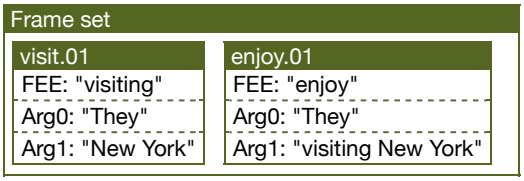

Figure 2: Semantic frames resulting from the comprehension process of the utterance "*They enjoy visiting New York.*" (COCA).

subsystem. We download a pretrained, precompiled grammar for English and load it into our agent using its `load_grammar_image` method. The agent now has at its disposal a grammar consisting of 21,052 constructions.

```
>>> pb_pretrained = fcg.PropBankAgent()
>>> f = fcg.load_resource('pb-en.store')
>>> pb_pretrained.load_grammar_image(f)
>>> pb_pretrained
<Agent: (id: agent-1) ~ 21052 cxns>
```

Our agent can now use its pretrained grammar to comprehend new utterances. Below, we instruct our agent to comprehend the passive utterance "*Margaret Thatcher was elected Prime Minister of Britain.*" (Herbst and Hoffmann, 2024, from NOW-19-12-08-US). The resulting meaning representation reveals that the agent identified a single semantic frame that instantiates the `elect.01` PropBank roleset ('*elect someone to an office or position*'). The agent also understood that the roles of '*candidate*' (`arg1`) and '*office or position*' (`arg2`) in this instance of `elect.01` are respectively taken up by "*Margaret Thatcher*" and "*Prime Minister of Britain*".

```
>>> pb_pretrained.comprehend("Margaret
    Thatcher was elected Prime Minister
    of Britain.")
[{'roleset': 'elect.01', 'roles': [
  ('v', "elected"),
  ('arg1', "Margaret Thatcher"),
  ('arg2', "Prime Minister of Britain")]
 }]
```

To enhance human readability, we can again choose to activate an FCG monitor to trace the comprehension process in the web interface. Figure 2 shows the visualisation of the two frames extracted from the utterance "*They enjoy visiting New York*" (COCA).

Let us now create a second agent, again as an instance of the `fcg.PropBankAgent` class, but let it learn a new grammar from corpus data instead of loading a pretrained one. After having downloaded an example CoNNL file, in which a number of English sentences are annotated with PropBank rolesets[8], we call the agent's `learn_grammar_from_conll_file` method. This call initiates the learning process implemented by the `fcg-propbank` subsystem and equips the agent with the resulting grammar. In this case, the agent has learnt two lexical constructions (for verbs with the lemmas *give* and *send*), two word sense constructions (for the rolesets `give.01` and `send.01`), and two argument structure constructions (a double object construction and a prepositional dative construction).

```
>>> pb_learner = fcg.PropBankAgent()
>>> f = fcg.load_resource('pb-
    annotations.conll')
>>> pb_learner.
    learn_grammar_from_conll_file(f)
>>> pb_learner
<Agent: (id: agent-2) ~ 6 cxns>
>>> list(pb_learner.grammar.cxns.keys())
['give(v)-cxn', 'send(v)-cxn',
 'give.01-cxn', 'send.01-cxn',
 'arg0(np)+v(v)+arg2(np)+arg1(np)-cxn',
 'arg0(np)+v(v)+arg1(np)+arg2(pp)-cxn']
```

We can now instruct our agent to comprehend a previously unseen utterance, using the grammar it just learnt, by calling its `comprehend` method. While comprehending "*The King of the Belgians sent a box of chocolates to Forrest Gump.*", the agent identifies an instance of the `send.01` ('*give*') roleset, with "*The King of the Belgians*" as the '*sender*' (`arg0`), "*a box of chocolates*" as the '*thing sent*' (`arg1`) and "*to Forrest Gump*" as the '*sent-to*' entity (`arg2`).

```
>>> pb_learner.comprehend("The King of
    the Belgians sent a box of
    chocolates to Forrest Gump.")
[{'roleset': 'send.01',
  'roles': [('v', "sent"),
   ('arg0', "The King of the Belgians"),
   ('arg1', "a box of chocolates"),
   ('arg2', "to Forrest Gump")]}]
```

After learning a grammar, it can be saved by calling the `save_grammar_image` method of the `fcg.Agent` class. This method saves the grammar to a file in a compiled, binary format that can later efficiently be loaded using an agent's `load_grammar_image` method.

```
>>> propbank_agent.save_grammar_image('
    usage-based-grammar.store')
>>> new_agent = fcg.PropBankAgent()
>>> new_agent.load_grammar_image('usage-
    based-grammar.store')
```

[8]Due to licensing restrictions, we are not able to provide large PropBank-annotated corpora as downloadable PyFCG resources. We invite interested readers to obtain such corpora (e.g. OntoNotes or EWT) directly from the *Linguistic Data Consortium*.

```
<Agent: (id: agent-3) ~ 6 cxns>
```

4.3 Modelling emergent communication

The final walkthrough tutorial exemplifies the primary use case of FCG: implementing the linguistic capability of autonomous agents in agent-based models of emergent communication. In such experiments, agents start out with an empty grammar and gradually build up their linguistic knowledge as they take part in situated communicative interactions with other agents in the population. This tutorial presents a PyFCG-powered implementation of the canonical naming game experiment (Steels, 1996; Van Eecke et al., 2022), in which a population of agents converges on a naming convention used to refer to objects in their environment. The choice for the naming game was made for didactic reasons, as implementations of language games involving the emergence of more complex grammars, where the use of a framework like FCG truly comes to its own, soon become strenuous to read through.

A first step in setting up a language game experiment concerns the creation of a population of agents. We define our agents as instances of a new class `NGAgent` that subclasses from PyFCG's `fcg.Agent` class. The agents are thereby initialised with an empty grammar and inherit a collection of methods for interacting with instances of the `fcg.Grammar` and `fcg.Construction` classes.

```
class NGAgent(fcg.Agent):
  ...
>>> NGAgent()
<Agent: agent (id: agent-1) ~ 0 cxns>
```

We also define a new experiment class `NGExperiment`. Upon initialisation, a population is created as a set of `NGAgent` instances, and a world is created as a set of abstract objects. Two methods are also associated to this class. The `run_interaction` method (cf. below) initiates a new communicative interaction as an instance of the `NGInteraction` class, makes the interaction happen, and records its outcome. The `run_series` method runs a given number of interactions.

```
class NGExperiment():
 def __init__(self, configuration={}):
  ...
  self.world =
  ['obj-%d' % i for i in range(
    configuration['nr_of_objects'])]
  self.population =
  [NGAgent() for i in range(
    configuration['nr_of_agents'])]

 def run_interaction(self):
  ci = NGInteraction(self)
  ci.interact()
  ci.record_communicative_success()
  ci.record_lexicon_size()
  ci.record_conventionality()

def run_series(self, nr_interactions):
 for i in range(nr_interactions):
  self.run_interaction()
```

The `interact` method of the `NGInteraction` class defines the script according to which each communicative interaction takes place. A randomly selected agent, the speaker, formulates an utterance to draw the attention of another randomly selected agent, the hearer, to a randomly selected object in the environment, the topic. If there exists no construction in the speaker's grammar that associates a name with the topic object, the speaker calls its `learn` method to invent such a construction. The hearer then calls its `comprehend` method to retrieve the topic object in the environment, and its `learn` method in case it could not understand. The agents achieve communicative success if the hearer could identify the topic object, and both agents will positively or negatively reward their constructions at the end of the interaction, based on its outcome.

```
class NGInteraction():
 ...
 def interact(self):
  s = self.speaker
  h = self.hearer
  s.utterance = s.formulate(s.topic)
  if s.utterance is None:
   s.learn(fcg.generate_word_form(), s.
    topic)
  if h.comprehend(s.utterance) is None:
   h.learn(s.utterance, s.topic)
  else:
   s.communicated_successfully = True
   h.communicated_successfully = True
  for agent in self.interacting_agents:
   agent.reward()
```

The `comprehend`, `formulate` and `reward` methods, as implemented for the NGAgent, illustrate how high-level PyFCG functionality facilitates the implementation of language game experiments. Not only do `comprehend` and `formulate` return the highest-scored solution, they also yield all competing solutions as a second return value. Successfully used constructions can then be rewarded positively through calls to their `increase_score` method and their competitors can be rewarded negatively through calls to their `decrease_score` method. Constructions that reach a score of 0 can be deleted from an agent's grammar using the `delete_cxn` method.

```
def reward(self):
```

```
 if self.communicated_successfully:
  self.applied_cxn.increase_score(0.1)
  for cxn in self.competitor_cxns:
   cxn.decrease_score(0.2)
   if cxn.get_score() <= 0.0:
    self.delete_cxn(cxn)
 else:
  if self.discourse_role == 'speaker':
   self.applied_cxn.decrease_score(0.2)
   if self.applied_cxn.get_score() <=
    0.0:
    self.delete_cxn(self.applied_cxn)
```

An experiment can be run by first creating a new instance of the `NGExperiment` class and then calling its `run_series` method, passing the desired number of interactions as an argument:

```
>>> ng = NGExperiment({'nr_of_agents':
    10, 'nr_of_objects': 5})
>>> ng.run_series(1500)
1500/1500 [100%] - 33.7s (44.50/s)
```

The results of an experiment can be visualised using any plotting library from Python's extensive ecosystem, such as `matplotlib`, `seaborn` or `plotly`. We demonstrate here the use of `matplotlib` to visualise the experiment's dynamics through graphs, where the degree of communicative success, degree of conventionality and average number of constructions are plotted in function of the number of interactions that have taken place (see e.g. Van Eecke et al., 2022).

```
import matplotlib.pyplot as plt
pp = make_plot_points(MONITORS)
fig, axes = plt.subplots(len(pp.keys()))
for i, key in enumerate(pp.keys()):
    ax = axes[i]
    ax.plot(list(range(len(pp[key]))),
       pp[key], label=key)
    ax.grid()
    ax.legend()
>>> plt.show()
```

Running this code yields the graphs presented in Figure 3, which show that the population indeed converges on a naming convention with one construction for each object in the world.

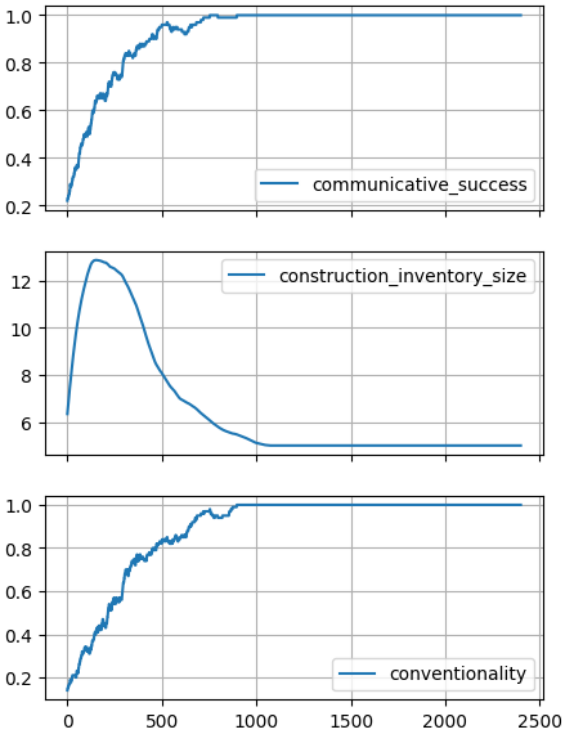

Figure 3: Graphs showing the dynamics of a single run of the PyFCG-powered naming game experiment, featuring 10 agents and 5 objects, created using the matplotlib library.

5 Technical Implementation

The design and technical implementation of PyFCG has been steered by two main considerations. First and foremost, the library needed to feel 'native' to Python users, rather than familiar to users already accustomed to the reference FCG implementation. Second, the library needed to wrap the reference implementation, rather than reimplement the original codebase.

A crucial aspect contributing to the 'native' look-and-feel of a Python library revolves around the library's embedding in the Python ecosystem of development tools. PyFCG is implemented as a Python package that is distributed via the Python Package Index (PyPI), and is hence '*pip-installable*' on (at least) macOS, Linux and Windows. PyFCG's codebase is versioned using Git and distributed under an open source license, with its documentation being available via the *Read the Docs* platform. Once installed, PyFCG exposes a range of high-level functions, classes and methods that build on common Python data structures such as objects, dictionaries, lists and tuples, thereby providing a truly '*Pythonic*' interface to FCG and ensuring maximal compatibility with other Python libraries.

The choice to wrap the reference implementation rather than providing a reimplementation of FCG was motivated by two main reasons. First of all, by wrapping the original Common Lisp implementation, PyFCG can leverage its optimised, multi-threaded codebase and achieve comparable efficiency when running FCG's most compute-intensive procedures. The second reason was to avoid distributing development and maintenance efforts over two distinct FCG implementations, which could easily become counterproductive in

the longer term.

Technically, PyFCG provides a bridge to a stand-alone, executable version of FCG, named *FCG Go*[9]. Upon calling PyFCG's `init` function, FCG Go is downloaded for the correct platform (if necessary) and launched as a background process. PyFCG communicates with this background process through HTTP requests, and thereby has access to the complete API of the reference implementation, as well as to its compiled codebase. When instances of PyFCG's `Grammar` and `Construction` classes are created, corresponding objects are instantiated in the FCG Go subprocess. Calls to methods that modify objects of these classes, such as `Agent.add_cxn` and `Construction.set_score`, modify their Python representation at the PyFCG side as well as their Common Lisp representation at the FCG Go side. Calls to functions and methods that more generally rely on functionality implemented by the FCG reference implementation, such as `fcg.start_web_interface` and `Agent.comprehend` are essentially rerouted to the running subprocess. Finally, functions and methods that do not modify FCG objects and are not compute-intensive, including those for listing an agent's constructions, for retrieving the scores of constructions, and for inspecting their internal representations, only involve objects on the Python side.

Crucially, the architecture of PyFCG ensures that its users do not notice that the library interfaces with a non-Python subprocess. The subprocess is automatically launched when the library is initiated, without the need to install a Common Lisp environment or any other dependencies. PyFCG's public interface is defined in terms of Pythonic data structures, any errors that would arise in the Common Lisp subprocess are caught and transposed to Python `Exceptions` (subclass `FcgError`), and the subprocess itself is safely closed by a clean-up method when Python exits.

6 Conclusion

This paper has presented PyFCG as an open source software library that ports Fluid Construction Grammar to the Python programming language. PyFCG is publicly distributed as a pip-installable Python package, and provides a fully Pythonic interface to FCG's reference implementation. The library thereby enables its users to seamlessly integrate constructional language processing and learning into Python programs, and to combine this functionality with that of other libraries within Python's extensive ecosystem. At the same time, the design of PyFCG as a wrapper around an executable version of FCG's Common Lisp reference implementation, which stealthily runs its multi-threaded and compute-intensive procedures in the background, ensures that PyFCG can incorporate all FCG functionality and that the community's development and maintenance efforts can remain united.

Apart from a general description of the library, its motivation and its technical implementation, this paper has presented three walkthrough tutorials that showcase how PyFCG can be integrated in typical use cases of FCG. The first tutorial has demonstrated how FCG agents can be created, how they can be equipped with a human-designed grammar, how they can be instructed to comprehend and formulate natural language utterances, and how these processes can be visually inspected using FCG's web interface. The second tutorial has showcased how FCG agents can learn grammars from semantically annotated corpora, how they can use these grammars to annotate new data, and how grammars consisting of tens of thousands of constructions can efficiently be saved and later reloaded into agents. The final tutorial has demonstrated the use of PyFCG in setting up agent-based experiments on emergent communication. We have taken the example of the canonical naming game and shown how high-level FCG functionality can ease the implementation of the language processing and learning capacities of individual agents.

As we reach the end of this paper, it is useful to remind ourselves that Fluid Construction Grammar was conceived as an open instrument that provides a collection of high-level building blocks for constructional language processing, with the goal of supporting research and applications at the intersection of construction grammar and computational linguistics. The walkthrough tutorials described in this paper, along with their accompanying interactive notebooks, demonstrate example usage of the library, but are not meant to serve as guidelines or rule book in any way. Fluid Construction Grammar has always evolved to fit the needs of its users, and new perspectives or use cases brought by PyFCG users or contributors will be heartily embraced.

[9] https://gitlab.ai.vub.ac.be/ehai/fcg-go

Acknowledgements

We would like to thank our system administrator Frederik Himpe for his continuous, cross-platform technical and moral support, Arno Temmerman for his help in releasing PyFCG through PyPI and *Read the Docs*, and Liesbet De Vos for designing the beautiful FCG Go logo.

References

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract meaning representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186.

Katrien Beuls and Paul Van Eecke. 2023. Fluid Construction Grammar: State of the art and future outlook. In *Proceedings of the First International Workshop on Construction Grammars and NLP (CxGs+NLP, GURT/SyntaxFest 2023)*, pages 41–50. Association for Computational Linguistics.

Katrien Beuls and Paul Van Eecke. 2024. Humans learn language from situated communicative interactions. What about machines? *Computational Linguistics*, 50(4):1277–1311.

Paul Boersma and David Weenink. 2025. Praat: doing phonetics by computer [Computer program]. Version 6.4.27, retrieved 27 January 2025.

Claire Bonial and Harish Tayyar Madabushi, editors. 2023. *Proceedings of the First International Workshop on Construction Grammars and NLP (CxGs+NLP, GURT/SyntaxFest 2023)*. Association for Computational Linguistics.

Claire Bonial and Harish Tayyar Madabushi. 2024. Constructing understanding: on the constructional information encoded in large language models. *Language Resources and Evaluation*.

William Croft. 2001. *Radical construction grammar: Syntactic theory in typological perspective*. Oxford University Press, Oxford, United Kingdom.

Charles J. Fillmore. 1988. The mechanisms of "construction grammar". In *Annual Meeting of the Berkeley Linguistics Society*, volume 14, pages 35–55.

Kateryna Gerasymova. 2012. Expressing grammatical meaning with morphology: A case study for russian aspect. In Luc Steels, editor, *Computational Issues in Fluid Construction Grammar*, volume 7249 of *Lecture Notes in Computer Science*, pages 91–122. Springer, Berlin, Germany.

Adele E. Goldberg. 2003. Constructions: A new theoretical approach to language. *Trends in Cognitive Sciences*, 7(5):219–224.

Thomas Herbst and Thomas Hoffmann. 2024. *A Construction Grammar of the English Language: CASA - a Constructionist Approach to Syntactic Analysis*. John Benjamins Publishing Company, Amsterdam/Philadelphia.

Thomas Hoffmann. 2018. Creativity and construction grammar: Cognitive and psychological issues. *Zeitschrift für Anglistik und Amerikanistik*, 66(3):259–276.

Yannick Jadoul, Bill Thompson, and Bart de Boer. 2018. Introducing Parselmouth: A Python interface to Praat. *Journal of Phonetics*, 71:1–15.

Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. 2014. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60.

Vanessa Micelli. 2012. Field topology and information structure: A case study for German Constituent Order. In Luc Steels, editor, *Computational Issues in Fluid Construction Grammar*, volume 7249 of *Lecture Notes in Computer Science*, pages 178–211. Springer, Berlin, Germany.

Thomas Moerman, Paul Van Eecke, and Katrien Beuls. 2024. Evaluating large-scale construction grammars on the tasks of semantic frame extraction and semantic role labeling. *Constructions*, 16(1):1–46.

Martha Palmer, Daniel Gildea, and Paul Kingsbury. 2005. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1):71–106.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, and 2 others. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems*, volume 32, pages 8026–8037. Curran Associates, Inc.

Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D. Manning. 2020. Stanza: A python natural language processing toolkit for many human languages. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 101–108.

Luc Steels. 1996. Perceptually grounded meaning creation. In *Proceedings of the Second International Conference on Multi-Agent Systems*, pages 338–344, Washington, D.C., USA. AAAI Press.

Luc Steels. 2004. Constructivist development of grounded construction grammar. In *Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL-04)*, pages 9–16.

Harish Tayyar Madabushi, Laurence Romain, Dagmar Divjak, and Petar Milin. 2020. CxGBERT: BERT meets construction grammar. In *Proceedings of the 28th International Conference on Computational Linguistics*, pages 4020–4032. International Committee on Computational Linguistics.

Yu-Hsiang Tseng, Cing-Fang Shih, Pin-Er Chen, Hsin-Yu Chou, Mao-Chang Ku, and Shu-Kai Hsieh. 2022. CxLM: A construction and context-aware language model. In *Proceedings of the Thirteenth Language Resources and Evaluation Conference*, pages 6361–6369.

Paul Van Eecke. 2018. *Generalisation and specialisation operators for computational construction grammar and their application in evolutionary linguistics Research*. Ph.D. thesis, Vrije Universiteit Brussel, Brussels: VUB Press.

Paul Van Eecke, Katrien Beuls, Jérôme Botoko Ekila, and Roxana Rădulescu. 2022. Language games meet multi-agent reinforcement learning: A case study for the naming game. *Journal of Language Evolution*, 7(2):213–223.

Remi van Trijp. 2016. *The evolution of case grammar*. Language Science Press, Berlin, Germany.

Remi van Trijp, Katrien Beuls, and Paul Van Eecke. 2022. The FCG Editor: An innovative environment for engineering computational construction grammars. *PLOS ONE*, 17(6):e0269708.

Leonie Weissweiler, Nina Böbel, Kirian Guiller, Santiago Herrera, Wesley Scivetti, Arthur Lorenzi, Nurit Melnik, Archna Bhatia, Hinrich Schütze, Lori Levin, Amir Zeldes, Joakim Nivre, William Croft, and Nathan Schneider. 2024. UCxn: Typologically informed annotation of constructions atop Universal Dependencies. In *Proceedings of the 2024 Joint International Conference on Computational Linguistics, Language Resources and Evaluation (LREC-COLING 2024)*, pages 16919–16932.

Leonie Weissweiler, Valentin Hofmann, Abdullatif Köksal, and Hinrich Schütze. 2022. The better your syntax, the better your semantics? Probing pretrained language models for the English comparative correlative. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 10859–10882, Abu Dhabi.

Lvxiaowei Xu, Zhilin Gong, Jianhua Dai, Tianxiang Wang, Ming Cai, and Jiawei Peng. 2024. CoELM: Construction-enhanced language modeling. In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 10061–10081.