Notice

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

Beuls Katrien, Van Trijp Remi and Wellens Pieter (2012). Diagnostics and Repairs in Fluid Construction Grammar. In Luc Steels and Manfred Hild (Eds.), *Language Grounding in Robots*, 215–234. Berlin: Springer.

BibTeX:

```
@incollection{beulsdiagnostics2012,
   Author = {Beuls, Katrien and Van Trijp, Remi and Wellens, Pieter},
   Title = {Diagnostics and Repairs in {Fluid Construction Grammar}},
   Pages = {215--234},
   Editor = {Steels, Luc},
   Booktitle = {Language Grounding in Robots},
   Publisher = {Springer},
   Address = {Berlin},
   Year = {2012}}
```

Chapter 1
Diagnostics and Repairs in Fluid Construction Grammar

Katrien Beuls[1], Remi van Trijp[2], and Pieter Wellens[1]

Abstract Linguistic utterances are full of errors and novel expressions, yet linguistic communication is remarkably robust. This paper presents a double-layered architecture for open-ended language processing, in which 'diagnostics' and 'repairs' operate on a meta-level for detecting and solving problems that may occur during habitual processing on a routine layer. Through concrete operational examples, this paper demonstrates how such an architecture can directly monitor and steer linguistic processing, and how language can be embedded in a larger cognitive system.

Key words: Fluid Construction Grammar, language processing, robustness

1.1 Introduction

Language users do not follow a rule book. Especially in spoken dialog, utterances are full of errors (such as hesitations, false starts and disconnected phrases) and novel expressions (such as word play, new or borrowed words and other innovations). Consider the following conversation between a foreign exchange student and the father of an English host family at the dinner table:

Example 1.
- Father: Could you pass me the salmon, please?
  (The student hesitates and then reaches for the salt.)
  (The father shakes his head.)
- Father: No, I meant the *salmon*. (Points to the fish on a plate.)
  (The student puts the salt back and hands over the plate.)
- Father: Thank you.

In this short interaction, several problems occur, which are solved in different ways using different sources of information. First, the student experiences difficul-

[1]VUB AI-Lab, Vrije Universiteit Brussel, e-mail: katrien@arti.vub.ac.be
[2]Sony Computer Science Laboratory Paris

ties in parsing the word *salmon*, but remembers the similar word *salt*, which happens to be a good fit in the current context. The father of the host family, however, sees that his utterance did not reach the desired effect and shakes his head to signal communicative failure. Knowing that the student does not yet fully master the English language, he therefore repeats the word *salmon* with more emphasis while pointing at the fish. The student now realizes that he in fact encountered a new word and tries to infer its meaning from the context.

The interaction is but one of the many illustrations that show that language is an inferential coding system (Sperber and Wilson, 1986) in which not all information is explicit in the message, but in which the listener is assumed to be intelligent enough to fill in the missing blanks. As Ronald Langacker (2000, p. 9) puts it:

> It is not the linguistic system *per se* that constructs and understands novel expressions, but rather the language user, who marshals for this purpose the full panoply of available resources. In addition to linguistic units, these resources include factors such as memory, planning, problem-solving ability, general knowledge, short- and longer-term goals, as well as full apprehension of the physical, social, cultural, and linguistic context.

The open-ended nature of language has caused many headaches to anyone who has ever attempted to implement language computationally because formalizations often seem to be too rigid and mechanical. One way to overcome such issues, as illustrated by Steels and van Trijp (2012), is to implement *diagnostics* for detecting problems in linguistic processing and *repairs* that solve those problems. This paper presents concrete examples of how this approach can be implemented in the *meta-level architecture* that forms an integral part of Fluid Construction Grammar (FCG; see Steels et al, 2012a in this volume and Steels, 2012a,c) and Babel (Steels and Loetzsch, 2010), a general cognitive framework that is used in the whole systems experiments discussed in other chapters of this book (Gerasymova and Spranger, 2012; Spranger et al, 2012; Spranger and Pauw, 2012; Steels, 2012d; Steels et al, 2012b). The architecture enables the grammar designer to build robust and open-ended grammars embedded in a larger cognitive system.

1.2 Situated Interactions

The sentence *No, I meant the salmon* and the corresponding pointing gesture of Example 1 only make sense as part of a situated dialog, not when studied in isolation. This paper therefore adopts the *language game methodology*, as introduced in an earlier chapter (Steels, 2012d). A language game can be considered as a microworld that operationalizes everything needed for modeling a routinized, communicative interaction: a situated context, two (or more) interlocutors, a communicative purpose, and so on. By grounding the speech participants in concrete communicative settings, language games allow pragmatic factors to play an important role as well.

Figure 1.1 illustrates the flow of a language game that roughly corresponds to Example 1. Here, the speaker asks for a certain object (such as the salmon). As a response, the listener can signal failure if he did not understand the question, or

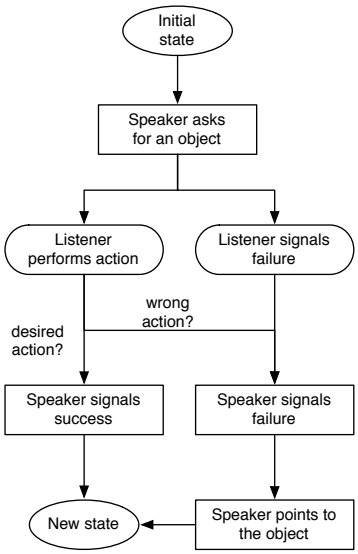

Fig. 1.1 Language games provide a way of modeling situated dialog. This diagram shows the possible flow of one game.

perform an action. If the listener signals failure or if he did not perform the action desired by the speaker, the speaker can provide feedback by pointing to the object he had in mind. If the hearer performed the action that the speaker was expecting, the game succeeds. There are many possible variations on this particular game (as is the case for Example 1) and there are many other kinds of games that can be played.

None of the nodes in Figure 1.1 are simple tasks, but each node can be broken down into several processes that correspond to different steps in the *semiotic cycle*, as illustrated by Figure 1.2. The semiotic cycle outlines the main steps that speakers and listeners have to go through when verbalizing and comprehending utterances as part of the language game. For instance, both speaker and listener need to build a *situation model* in which they maintain a connection between their internal factual memory and the states and actions in the world. The speaker (shown on the left in Figure 1.2) then needs to decide on a communicative *goal* (such as obtaining the salmon) and *conceptualize* a meaning in such a way that it satisfies his communicative goal when expressed through language (*production*). The hearer (shown on the right in the Figure) needs to *parse* the observed utterance in order to reconstruct its meaning and then *interpret* that meaning into his situation model, where he confronts it with his appreciation of the context and his own factual memory. If the hearer successfully retrieves the speaker's intended goal, he may *act* accordingly. In Example 1, however, the listener's action did not correspond to the desired one, so the interlocutors have to go through the semiotic cycle again.

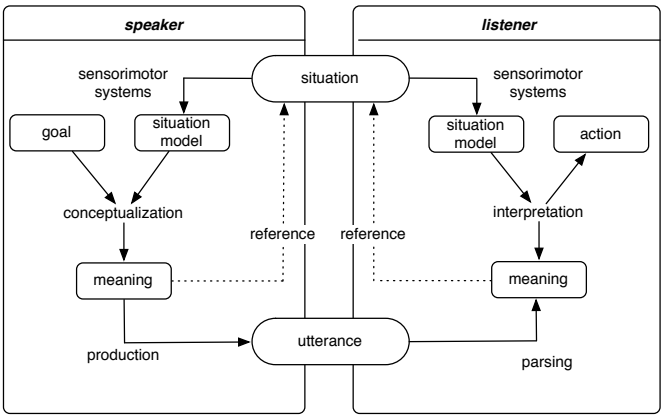

Fig. 1.2 The semiotic cycle summarizes the main processes that speakers (left) and listeners (right) go through when playing language games.

Each process in the semiotic cycle can in turn be dissected into smaller steps. In this paper we are mainly concerned with *production* and *parsing*, which are the linguistic processes handled by Fluid Construction Grammar. As already introduced in an earlier chapter in this book (Steels et al, 2012a) and explained in more detail by Steels (2012b); Bleys et al (2012), the FCG-interpreter handles linguistic processing as a *search problem* in which the appropriate set of constructions need to be found that, when applied, succeed in verbalizing a particular meaning (production) or analyzing an observed utterance (parsing).

1.3 A Meta-level Architecture for Problem Solving

The architecture of FCG and Babel has a double-layered design, as shown in Figure 1.3. The first layer is called the *routine layer* and handles habitual processing. A second layer, called the *meta-layer*, monitors and sometimes steers routine processing through *diagnostics* and *repairs*, which try to detect and solve problems that may occur in the routine layer. Repairs have the power to modify an agent's inventory of concepts, linguistic constructions, beliefs, and so on. They can also go back a few steps, for instance choosing a different communicative goal or parsing an utterance again, in order to test whether the repair adequately solves the detected problem. Problems can be detected at each level, at each step and at any time; and different repairs can be triggered in succession of each other. This constant interaction between routine- and meta-layer processing ensures robustness and open-endedness for coping with noise or variation in perception, differences in embodiment, novelty, and other problems that inevitably occur in linguistic interactions.

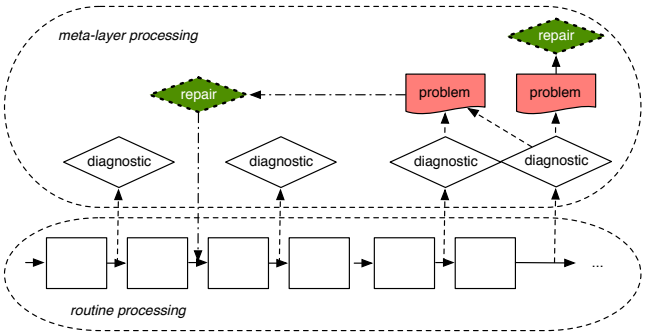

Fig. 1.3 FCG and Babel feature a double-layered architecture. Besides a routine layer for habitual processing, a meta-layer of diagnostics and repairs detect and solve problems that may occur.

1.3.1 Basic Definitions

This section offers some basic definitions that build on earlier work on meta-layer architectures for multi-agent modeling (Steels and Loetzsch, 2010). The standard architecture uses an object-oriented approach and has been implemented in CLOS (Common Lisp Object System; see Keene, 1988). It builds on three main classes that are represented by the boxes in the meta-layer in Figure 1.3: `problem`, `diagnostic` and `repair strategy`. Specific implementations of diagnostics and repairs subclass from these general classes and add further semantics. The three base classes are defined as follows:

Definition 1.

class	**problem**
Description:	*The base class for all problems. A* `problem` *is instantiated by a* `diagnostic` *to report a failure or some inefficiency.*
Slots:	`issued-after` `repaired-by`

A `problem` has two slots: `issued-after` and `repaired-by`. The first slot can be filled by a symbol that specifies when the problem has been instantiated and reported (for instance, after a *production* process). If the problem has been solved, the value of the second slot is automatically set to the name of the `repair-strategy` that repaired it. If not, its value is set to the empty list `nil`.

Definition 2.

class	**diagnostic**
Description	*The base class for all* diagnostics. *Diagnostics instantiate a* `problem` *if they detect a failure or inefficiency.*
Slots	`learning-situations`

Diagnostics are responsible for finding difficulties and instantiating a `problem` for reporting them. The only slot of the base class (`learning-situations`) is used for specifying in which situations the diagnostic is active. For instance, some diagnostics only need to be executing when acting as the listener.

Definition 3.

class	**repair-strategy**
Description:	*The base class for all repairs. Repair strategies handle* problems.

Slots:	`triggered-by-problems`
	`learning-situations`
	`success-score`

A `repair-strategy` has three slots. `Triggered-by-problems`, the first slot, contains the names of problems that trigger the activation of the repair strategy. The second slot `learning-situations` narrows down the point of execution of a repair strategy similar to the same slot in the base class for diagnostics. The kinds of learning situations depends on the level the repair strategy is operating on. The `success-score` reflects how successful the repair-strategy has been in solving previous problems. If a problem has been reported that can be solved by multiple repair strategies, the repair with the highest success-score is tried first.

1.3.2 Three Levels of Application

In line with the language game approach (see Section 1.2), we address language as a problem-solving activity on (at least) three levels:

1. The *FCG-level*, which concerns linguistic processing itself whereby the FCG-interpreter needs to parse and produce utterances.
2. The *Process-level*, which corresponds to cognitive processes in the semiotic cycle (see Figure 1.2).
3. The *Agent-level*, which covers behaviors and turn-taking in a language game (see Figure 1.1).

The FCG-level is embedded within the other levels through the general cognitive framework Babel (Steels and Loetzsch, 2010). At each level, an *agent* (which models a language user), performs problem-solving activities for achieving communicative goals and subgoals. When speakers make errors or need to use novel expressions, however, an agent's current state (including his knowledge, beliefs, and so on) may not suffice for finding adequate solutions. Every level has its own classes and methods for defining diagnostics and repairs that all subclass from the basic definitions introduced in Section 1.3.1. The following subsections provide the technical details of these classes and methods, which can be used by the reader as background reference for understanding the examples of section 1.4.

1.3.2.1 FCG-Level Definitions

First, the `fcg-diagnostic` class is a subclass of `diagnostic`. It has one additional slot `direction`, whose value is either the symbol → (which stands for production) or ← (which stands for parsing):

Definition 4.

class	**fcg-diagnostic**	subclass of diagnostic
Description:	*A diagnostic that can be activated during parsing and production.*	
Slots:	`direction`	

Associated with the `fcg-diagnostic` is a *generic function*, which in CLOS "defines an abstract operation, specifying its name and a parameter list but no implementation" (Seibel, 2005, p. 191). The generic function `diagnose-fcg` has two parameters: an fcg-diagnostic and a node from an FCG search process:

Definition 5.

generic function	**diagnose-fcg**
Description:	*Can be called at each FCG search node.*
Parameters:	`fcg-diagnostic` `search-node`

For each FCG-diagnostic, it is thus necessary to write a method that actually executes the diagnostic. Methods "indicate what kinds of arguments they can handle by *specializing* the required parameters defined by the generic function" (Seibel, 2005, p. 192). For example, a method may specialize on a specific kind of node.

Next, FCG has its own class for repairs. An `fcg-repair-strategy` subclasses from `repair-strategy` and defines one additional slot that specifies whether the repair is called in production or parsing:

Definition 6.

class	**fcg-repair-strategy**	subclass of repair-strategy
Description:	*A repair that operates during parsing and production.*	
Slots:	`direction`	

Again, there is a generic function associated with FCG-level repairs, which defines three parameters: an FCG-level repair, a problem, and an FCG node. Each `fcg-repair-strategy` thus requires a method that specializes on these three parameters, for example a method that can handle an `unknown-word` problem.

Definition 7.

generic function	**repair-fcg**
Description:	*Is called a new FCG node has been created.*

Parameters:	`fcg-repair-strategy`
	`problem`
	`search-node`

1.3.2.2 Process-Level Definitions

Process diagnostics and repairs can be run after any given process. The class `process-diagnostic` has a single slot: `trigger-processes`. These are the names of processes after which this diagnostic should be triggered, such as `parse`, `conceptualize`, etc.

Definition 8.

class	**process-diagnostic**	subclass of diagnostic
Description:	*A diagnostic that is triggered after the execution of a process.*	
Slots:	`trigger-processes`	

To run a process diagnostic one has to implement a diagnose-process method. `diagnose-process` returns either one problem, a list of problems or `nil`. If one or more problems are returned they are automatically added to the problems of the current turn. When nil is returned, no problems were detected.

Definition 9.

generic function	**diagnose-process**
Description:	*Is called after running a process and handling its process results.*

Parameters:	`process-diagnostic`
	`turn`
	`process`

Also on the process level, a general repair strategy class has been implemented to host more specific repairs on this level. Process repair strategies try to repair problems in the current turn, which could also be problems created by lower-level diagnostics (i.e. in the FCG search).

Definition 10.

class	**process-repair-strategy**	subclass of repair-strategy
Description:	*A repair that can operate after the execution of a process.*	

Every process repair strategy requires a specialized `repair-process` method. `repair-process` returns two values: a first one to indicate its success (boolean) and a second one to signal a restart. The second value is the name of the process that must be restarted (e.g. `parse`). If the second value is nil processing will continue where it left off.

Definition 11.

generic function	**repair-process**
Description:	*Is called when problems occurred between processes.*

Parameters:	`process-repair-strategy`
	`problem`
	`turn`
	`process`

1.3.2.3 Agent-Level Meta-Operators

Sometimes it is impossible to diagnose or repair something in a single turn, for instance when the listener first requires feedback from the speaker before he can guess the meaning of a new word. For this reason, we support meta-operators on an even higher level: that of one agent. Again, a general `agent-diagnostic` class is available. It has no additional slots.

Definition 12.

class	**agent-diagnostic**	subclass of diagnostic
Description:	*A diagnostic that is triggered after an agent finished his turn.*	

After an agent has finished his turn (e.g. speaking), the `diagnose-agent` method is called for executing every agent-diagnostic that has been defined. This method returns one or more problems, or `nil`. The `agent-interaction-point` can be any point in a language game, such as `listening`, `speaking`, `pointing`, and so on.

Definition 13.

generic function	**diagnose-agent**
Description:	*Diagnose-agent is called for every agent-diagnostic when an agent finished his turn.*

Parameters:	`agent-diagnostic`
	`agent-interaction-point`
	`agent`
	`world`

Repair strategies on the Agent-level are defined in the `agent-repair-strategies` class. Agent repair strategies are executed as soon as one agent has finished his turn. They try to repair any detected problem, which could again also be problems created by lower-level diagnostics (i.e. FCG- or Process-level).

Definition 14.

class	**agent-repair-strategy**	subclass of repair-strategy
Description:	*A repair that is triggered after an agent finished his turn.*	

Specialized `repair-agent` methods need to be implemented for executing the repair strategies. These methods return two values: a boolean for indicating whether the repair was successfully executed or not, and a request to restart an agent's turn.

Definition 15.

generic function	**repair-agent**
Description:	*Repair-agent is called after an agent finished his turn. Might also repair problems of lower-level diagnostics.*

Parameters:	`agent-repair-strategy`
	`agent-interaction-point`
	`problem`
	`agent`
	`world`

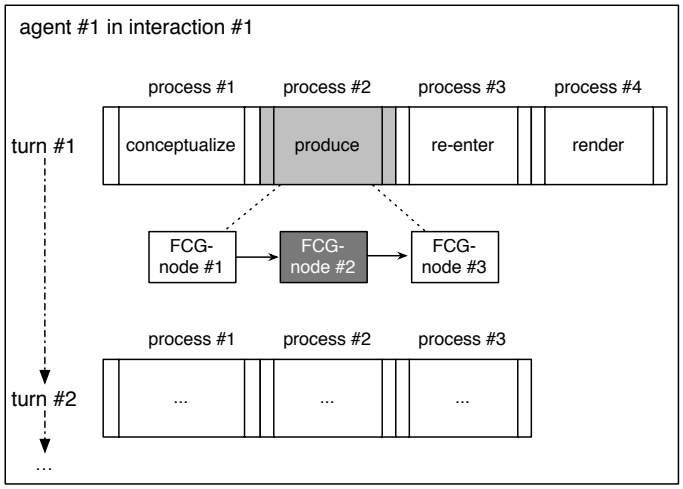

Fig. 1.4 The design of diagnostics and repairs is based on the principle of devolution.

1.3.3 Principle of Devolution

As a rule of thumb, the choice for implementing a diagnostic or repair on the agent-, process- or FCG-level should be based on the *principle of devolution*, which means that everything that can better be managed and decided 'on the spot', should be. Instead of always opting for a centralized, high-level approach, specific diagnostics and repairs (which are also called *meta-operators*) are therefore *devolved* to the particular level of the meta-level architecture where they are most efficient.

The main advantage of devolving meta-operators to specific levels is efficiency, as illustrated in Figure 1.4. On the highest level of information processing, the Agent-level meta-operators monitor and steer longer- and shorter-term discourse goals and turn-taking in the interaction. For example, an Agent-level diagnostic can detect whether the listener's response corresponds to the speaker's desired communicative goal. In principle, the operators can also detect whether any problems occurred within a particular turn in the language game, but they cannot directly intervene in the processes that try to achieve the subgoal of that turn (e.g. asking a question). All the Agent-level meta-operators can do is detect a problem with the output of those processes and then restart them again.

Problems that occur within a single turn are therefore better handled by Process-level meta-operators, which manage all the steps of the semiotic cycle (see Figure 1.2) that a speaker or listener needs to go through in order to verbalize or comprehend utterances. These operators are best suited for monitoring the information flow between different steps (for instance whether conceptualization has come up with a

meaning that can be expressed by the language) and the processing effort required for each step (for example how many possible interpretations can be found for an utterance). Like the Agent-level operators, however, Process-level operators cannot directly intervene within a particular step and only works on their output.

Meta-operators on the FCG-level, then, can be seen as the 'field workers' that directly act upon FCG's search in production and parsing, and hence are able to diagnose problems in linguistic processing as soon as they occur and possibly solve them. Similar process-internal meta-operators can be specialized for other steps in the semiotic cycle to improve efficiency, for example within conceptualization where a speaker has to plan what to say, but they are not handled by this paper.

Despite being defined on different levels, all diagnostics and repairs can nevertheless cooperate with each other because the `problem` class is level-independent. For example, an FCG-level diagnostic can detect an unknown word in parsing and then instantiate a `problem` in which information about the unknown word is passed to an Agent-level repair, which can then try to solve the problem by asking for feedback. By using problems as mediators between meta-operators instead of directly linking diagnostics to repairs, the experimenter has complete control over the way in which difficulties can be detected and solved.

1.3.4 Restart Requested

The basic unit on which the learning operators function is *a search node* (see Figure 1.3). This node can either be a complete process inside a turn of one agent or an FCG-node within one of the linguistic processes of the turn. Meta-operators are thus automatically passed to the appropriate level. Also the grammar of an agent is copied to the lower levels to provide the possibility to adapt it in repair.

When a problem is instantiated in a search node, it is always local to the branch it is detected in. This is important since other unexplored branches might not generate the same problem. A problem contains a pointer to the complete search tree so that the current best solution can constantly be updated, taking into account the problems that have been signaled on different branches. A solution is always the branch with the highest success score.

The option exists to restart the processing pipeline at a predefined node when a successful repair took place. Figure 1.5 illustrates this approach. Any of the ancestors of the problematic node could potentially have caused the problem. A successful repair automatically removes the repaired problem from the search tree. When a node is restarted, data belonging to this node can be overridden depending on the changes the repair made. For instance, when a new construction has been added to the grammatical inventory, the restart node should have access to the latest version of the grammar. It is important to note that when a restart is requested, all nodes that are descendants of the restart node are deleted. This means that potential solutions (in sibling branches) can be lost after a restart. It is therefore safer to verify whether the search process has been finished before the search is restarted.

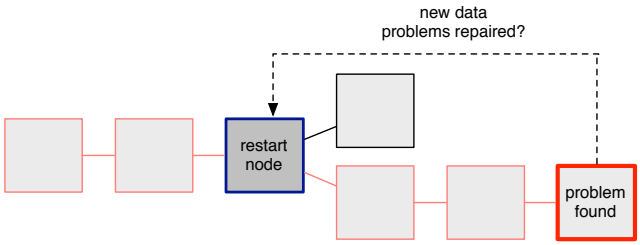

Fig. 1.5 A restart is requested after a problem has been found in a node. The search is restarted at the point where a split occurred.

1.4 The Salmon Game Revisited

The remainder of this paper presents concrete *use cases* of problems that may occur on the agent-, process- and FCG-level (with a specific focus on the latter level), and then provides operational examples of how diagnostics and repairs can be implemented for solving those problems. We do not claim any cognitive or psychological plausibility on the particular diagnostics and repairs that are described, but rather aim at illustrating how grammar engineers can use the meta-level architecture for designing their own solutions. Indeed, the architecture remains agnostic as to which sets of diagnostics and repairs are most adequate and plausible.

1.4.1 FCG-level

Diagnosing and repairing problems on the FCG-level is not only efficient because it allows problems to be detected and solved as they occur in processing; it also allows grammar engineers to define open-ended, standalone FCG grammars outside of the Babel framework.

1.4.1.1 Use Case

A widely known challenge for precision grammars is lexical coverage. For example, when testing the English Resource Grammar (the most complete computational formalization of English to date; Copestake and Flickinger, 2000) against a random sample of 20.000 strings from the British National Corpus, 41% of the parsing failures were caused by missing lexical entries (Baldwin et al, 2005). The meta-level architecture of Babel and FCG offers grammar engineers the necessary tools for exploring which solutions may overcome this problem.

Let us return to the salmon game that opened this paper. The exchange student did not understand what exactly he was supposed to hand over to his host father. He therefore made a guess and reached out for the salt, an object on the table whose name closely resembles that of the requested item. We can model this process of finding the closest match once an unknown word has been detected by means of the FCG-level meta-operators. This section illustrates how this can be done with an FCG-diagnostic and -repair. The general problem that glues these operators together is the `unknown-word` problem, which subclasses from `problem`. It contains one additional slot whose value contains the unknown word, which can then be passed to any repair strategy that tries to handle the problem:

Definition 16.

class	**unknown-word**	subclass of problem
Description:	*Instantiated when unprocessed words are diagnosed in the linguistic structure.*	
Slots:	`word`	

1.4.1.2 Diagnostic

How can we now implement a way for detecting unknown words? First, we define a new FCG-diagnostic and set the slot-value of its direction to ←, which means that it should be activated during parsing:

Definition 17.

class	**detect-unknown-words-in-fcg-search**	subclass of fcg-diagnostic
Description:	*Diagnoses unprocessed words in parsing.*	
Set slot-value:	`direction`	←

Now we can define a `diagnose-fcg` method that specializes on this new class. Here, we define a method that takes its second argument (i.e. an FCG node) and checks whether there are unprocessed strings left in the linguistic structure that is contained in the node. The method only cares about *leaf nodes*, which are the last nodes of the branches of a search tree, which means that no constructions can apply anymore. If there is one unknown string, the method instantiates an `unknown-word` problem. For illustration purposes, the diagnostic only handles single unknown words instead of multiple unknown strings. In pseudo code, the method looks as follows:

diagnose-fcg `(detect-unknown-word-in-fcg-search FCG-node)`

```
When NODE is a LEAF then:
  let UNPROCESSED-STRINGS be the EXTRACTED-UNPROCESSED-STRINGS of FCG-NODE
     if UNPROCESSED-STRINGS contains a SINGLE-WORD
     then return an instance of UNKNOWN-WORD
              and set the slot-value of :WORD to SINGLE-WORD
     else return NIL
```

In the FCG's interactive web interface (Loetzsch, 2012), problematic nodes are colored differently than successful ones, and they receive an additional status:

problem-found. Figure 1.6 shows a screen shot of such a node, where the 'top-unit' (the open box to the right) acts as a buffer that contains all unprocessed information. As can be seen, the unprocessed string as signaled by the diagnostic is "salmon". Also the word order conditions (cf. meets attributes) are still unprocessed at this stage.

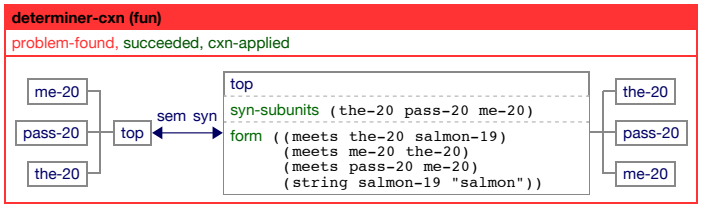

Fig. 1.6 A problem is diagnosed after the string "salmon" is left unprocessed at the end of the search tree.

1.4.1.3 Repair

Once an unknown word has been detected inside the FCG search tree, a repair will trigger and try to solve the problem. In the use case that we are investigating here, the unknown word is "salmon". An example of an FCG repair strategy that tackles this problem is retry-with-closest-match. Such a strategy loops through all words in the current grammar and find the word that mostly resembles the unknown word based on its form. The example repair strategy here only considers similarity in terms of spelling, not in phonetic form. In a more advanced implementation, the latter could of course also be taken into account.

The repair strategy is initialized with the following slot values:

Definition 18.

class	**retry-with-closest-match**	subclass of fcg-repair
Description:	*Repairs unprocessed words in parsing.*	
Set slot-value:	direction	←
Set slot-value:	triggered-by-problems	unknown-word

When these initial values are satisfied, a specialized repair-fcg method can execute this particular repair strategy. The pseudo code explains how the original utterance by the host father (expert-utterance) is replaced with a slightly modified version (learner-utterance) by substituting the unknown word with its closest match. The function find-closest-string is responsible for searching the existing lexical items and returning the most similar word.

repair-fcg `(retry-with-closest-match problem FCG-node)`

```
Let UTTERANCE be the RENDERED LINGUISTIC STRUCTURE of FCG-NODE
and UNKNOWN-WORD be the :WORD slot in PROBLEM
and CLOSEST-MATCH be the UNKNOWN-WORD'S CLOSEST RELATED WORD in LEXICON
   if there is a CLOSEST-MATCH
   then return TRUE
      and let the REVISED-UTTERANCE be the UTTERANCE after the UNKNOWN-WORD
              has been REPLACED with CLOSEST-MATCH
       then RESTART SEARCH TREE with REVISED-UTTERANCE
   else return NIL
```

When the search tree is restarted, the initial node contains the substituted utterance (see Figure 1.7) and parsing succeeds.

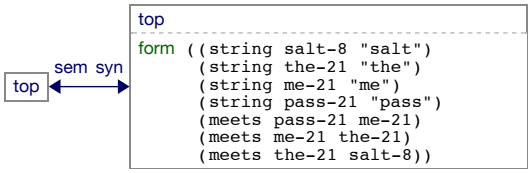

Fig. 1.7 The new initial node after processing has been restarted.

Although the processing problem has been repaired, the game still fails since the student did not manage to retrieve the correct object form the context. The student also did not really learn something, that is, in technical terms no new construction was added to the grammar. The following section illustrates how a construction can be added while repairing.

1.4.2 Process-level

Process-level learning operators allow the experimenter to diagnose and repair problems after each step in the semiotic cycle. In case of the exchange student, the steps (or processes) that have to be monitored are de-rendering, parsing and interpretation. Since the use case has remained the same (the salmon game), the problem that is diagnosed is still `unknown-word`. The following sections illustrate the use of Process-level operators for diagnosing and repairing this problem.

1.4.2.1 Diagnostic

First we define `detect-unknown-word-after-parse`, an instance of a process-diagnostic that is triggered by the process `'parse`, which means that the diagnostic needs to be executed after parsing the utterance:

Definition 19.

class	**detect-unknown-words-after-parse**	subclass of process-diagnostic
Description:	*Diagnoses unprocessed words after the parse process.*	
Set slot-value:	`trigger-processes`	*parse*
Set slot-value:	`learning-situations`	*listening*

The method that executes the diagnostic is similar to its FCG-variant in the sense that it extracts strings from a linguistic structure. The main difference lies in the object that is manipulated: instead of an FCG-node, the diagnostic takes a full process result (i.e. a parsing result) and the name of the agent's turn as its arguments. The unprocessed strings can be accessed by extracting them from the last FCG node of the linguistic process that is being diagnosed.

diagnose-process `(detect-unknown-word-after-parse turn process)`

```
Let UNPROCESSED-STRINGS be the EXTRACTED-UNPROCESSED-STRINGS
                                from the FINAL FCG-NODE in PROCESS
if UNPROCESSED-STRINGS contains a SINGLE-WORD
then return an instance of UNKNOWN-WORD
      and set the slot-value of :WORD to SINGLE-WORD
else return NIL
```

1.4.2.2 Repair

Here we define a process repair strategy (`add-generic-cxn`) that is triggered by the `unknown-word` problem:

Definition 20.

class	**add-generic-cxn**	subclass of fcg-repair
Description:	*Repairs unprocessed words in parsing.*	
Set slot-value:	`learning-situations`	listening
Set slot-value:	`triggered-by-problems`	unknown-word

A possible method for executing this repair strategy is to use a "generic" construction that takes the unknown word as its form, but which leaves its meaning and semantic and syntactic categorization underspecified. The pseudo code of the repair function looks as follows:

repair-process `(add-generic-cxn problem turn process)`

```
If there is an UNKNOWN-WORD in PROBLEM
then add a GENERIC CONSTRUCTION of UNKNOWN-WORD to TURN-GRAMMAR
   and return TRUE and restart process PARSE
else return NIL
```

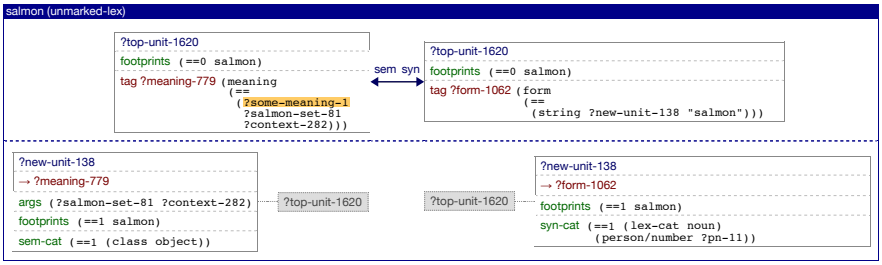

Fig. 1.8 The lexical construction that is added for the unknown word “salmon”. No specific meaning is added at the moment of creation, but this could be added later by another repair (e.g. agent repair after pointing; cf. infra).

In case of successful repair, the method requests a restart of the process `'parse`, using the generic construction as depicted in Figure 1.8. The generic construction allows the agent to parse the utterance, but its lack of a specific meaning prevents the listener of finding the object that was asked by the host father. Given the situation in which the salmon game is embedded, an alternative repair strategy is therefore to attribute a temporary meaning to the unknown word “salmon”. At the moment of diagnosing the `unknown-word` problem, the meaning that has been parsed so far indicates that the unknown word is the object of the passing event. In the context of a family dinner, the listener could thus infer that the requested object probably meets the following semantic conditions: `(edible +)` `(definite +)` and `(graspable +)`. In this sense, the grammar could be searched for objects that fulfill these conditions and (optionally) share a similar word form. In line with the FCG repair strategy, a construction could then be added that maps the “salmon" word form to the `salt` meaning predicate:

$$\texttt{salt} \Longleftrightarrow \texttt{"salmon"}$$

If this construction would be used in parsing, the student would again reach for the salt and later receive the information that the father meant the plate with salmon. The final level in the architecture will allow us to incorporate this information and learn the correct mapping between meaning and form.

1.4.3 Agent-level

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.

This is exactly the case in the salmon game. When an agent participates in a language game he carries out multiple actions, such as speaking, listening, pointing, signaling an error, etc. In each of these actions, a problem can occur. But instead of instantiating a new problem and a new diagnostic for this level, we recycle the original `unknown-word` problem and the process-level diagnostic `detect-unknown-word-after-parse`. The compatibility of learning operators from different levels is a powerful feature of the meta-layer architecture. It allows the experimenter to diagnose a problem early on and wait to repair it until more information has become available. Of course, sometimes it is indispensable to add an additional agent-diagnostic to signal problems in the agent's actions themselves, such as in pointing to an object that cannot be retrieved from the situation.

The agent repair strategy specialized for the `unknown-word` problem triggers when the listener has perceived a pointing action:

Definition 21.

class	**adopt-new-cxn**	subclass of fcg-repair
Description:	*Repairs unprocessed words in parsing.*	
Set slot-value:	`learning-situations`	listener-perceives-pointing
Set slot-value:	`triggered-by-problems`	unknown-word

The most straightforward repair strategy that presents itself in the context of the salmon game is one that makes use of the object of the pointing action and couples it to the unknown word. This coupling is casted into a new construction that is added to the listener's grammar. In the salmon game, the mapping would be the following:

$$\texttt{salmon} \Longleftrightarrow \texttt{"salmon"}$$

Pseudocode for the main repair function that uses this information is included below.

repair-agent `(adopt-new-cxn agent-interaction-point problem agent world)`

```
If there is an UNKNOWN-WORD in PROBLEM
then add a LEXICAL CONSTRUCTION for UNKNOWN-WORD to AGENT-GRAMMAR
   and return TRUE
else return NIL
```

The game is not restarted here after the learner agent has adopted the new lexical construction. The next time the word "salmon" is parsed, the unknown word problem will not be diagnosed again.

1.5 Conclusion

This chapter has illustrated the workings of the meta-level computational layer that is present in the architecture of Fluid Construction Grammar (FCG) and, more

largely speaking, the Babel platform. The decomposition of computation in separate modules for routine language processing and meta-level computation guarantees the effective and smooth functioning of routine grammatical processing in FCG. By means of three constructive examples that all apply to the failed communication in the salmon game (see Example 1), every level of the meta-level architecture has been explained and illustrated, with specific attention for the tools that are available to program the meta layer operators.

Acknowledgements

This research was conducted at the VUB AI-Lab at the University of Brussels and at the Sony Computer Science Laboratory in Paris. Katrien Beuls received funding from a strategic basic research grant from the agency for Innovation by Science and Technology (IWT). Pieter Wellens has been supported by the ESF EuroUnderstanding project DRUST. Additional funding came from the FP6 EU project ECAgents and the FP7 EU project ALEAR. We would like to thank Luc Steels, director of the Paris and Brussels labs, for his support and feedback. We would also like to thank all members of our team for continuously reshaping the way the FCG architecture is constructed through their productive feedback on earlier versions of the implementation. Some parts of this document have been adapted from the Babel2 manual (Loetzsch et al, 2008). All remaining errors in the explanation of the architecture are of course our own. The latest release of Babel can be downloaded from: https://arti.vub.ac.be/trac/babel2.

References

Baldwin T, Beavers J, Bender EM, Flickinger D, Kim A, Oepen S (2005) Beauty and the beast: What running a broad-coverage precision grammar over the BNC taught us about the grammar – and the corpus. In: Kepser S, Reis M (eds) Linguistic Evidence: Empirical, Theoretical, and Computational Perspectives, Mouton de Gruyter, Berlin, pp 49–69

Bleys J, Stadler K, De Beule J (2012) Search in linguistic processing. In: Steels L (ed) Design Patterns in Fluid Construction Grammar, John Benjamins, Amsterdam

Copestake A, Flickinger D (2000) An open-source grammar development environment and broad-coverage English grammar using HPSG. In: Proceedings of the 2nd International Conference on Language Resources and Evaluation (LREC 2000), pp 591–600

Gerasymova K, Spranger M (2012) An Experiment in Temporal Language Learning. In: Steels L, Hild M (eds) Language Grounding in Robots, Springer, New York

Keene S (1988) Object-Oriented Programming in Common Lisp: A Programmar's Guide to CLOS. Addison-Wesley, Boston (MA)

Langacker RW (2000) A dynamic usage-based model. In: Barlow M, Kemmer S (eds) Usage-Based Models of Language, Chicago University Press, Chicago, pp 1–63

Loetzsch M (2012) Tools for grammar engineering. In: Steels L (ed) Computational Issues in Fluid Construction Grammar, Springer Verlag, Berlin

Loetzsch M, Wellens P, De Beule J, Bleys J, van Trijp R (2008) The babel2 manual. Tech. Rep. AI-Memo 01-08, AI-Lab VUB, Brussels

Seibel P (2005) Practical Common Lisp. Apress, Berkeley, CA

Sperber D, Wilson D (1986) Relevance: Communication and Cognition. Harvard University Press, Cambridge, MA

Spranger M, Pauw S (2012) Dealing with Perceptual Deviation: Vague Semantics for Spatial Language and Determiners. In: Steels L, Hild M (eds) Language Grounding in Robots, Springer, New York

Spranger M, Pauw S, Loetzsch M, Steels L (2012) Open-ended Procedural Semantics. In: Steels L, Hild M (eds) Language Grounding in Robots, Springer, New York

Steels L (ed) (2012a) Computational Issues in Fluid Construction Grammar. Springer Verlag, Berlin

Steels L (2012b) Design methods for Fluid Construction Grammar. In: Steels L (ed) Computational Issues in Fluid Construction Grammar, Springer Verlag, Berlin

Steels L (ed) (2012c) Design Patterns in Fluid Construction Grammar. John Benjamins, Amsterdam

Steels L (2012d) Grounding Language through Evolutionary Language Games. In: Steels L, Hild M (eds) Language Grounding in Robots, Springer, New York

Steels L, Loetzsch M (2010) 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, Springer Verlag, Berlin, pp 307–313

Steels L, van Trijp R (2012) How to make construction grammars fluid and robust. In: Steels L (ed) Design Patterns in Fluid Construction Grammar, John Benjamins, Amsterdam

Steels L, De Beule J, Wellens P (2012a) Fluid Construction Grammar on Real Robots. In: Steels L, Hild M (eds) Language Grounding in Robots, Springer, New York

Steels L, Spranger M, van Trijp R, Höfer S, Hild M (2012b) Emergent Action Language on Real Robots. In: Steels L, Hild M (eds) Language Grounding in Robots, Springer, New York