In: Proceedings of the Workshop "Computational Environments for Grammar Development and Linguistic Engineering (ENVGRAM)" held in conjunction with the 35th Meeting of the ACL and 8th Conference of the EACL. Madrid, Spain. 1997.

The ConTroll System as Large Grammar Development Platform

Thilo Götz and **Walt Detmar Meurers**[*]
SFB 340, Universität Tübingen
Kleine Wilhelmstraße 113
72074 Tübingen
Germany
{tg,dm}@sfs.nphil.uni-tuebingen.de

Abstract

ConTroll is a grammar development system which supports the implementation of current constraint-based theories. It uses strongly typed feature structures as its principal data structure and offers definite relations, universal constraints, and lexical rules to express grammar constraints. The aspects of ConTroll of relevance to the development of large grammars are discussed in detail. The system is fully implemented and has been used as workbench to develop and test a large HPSG grammar.

1 Introduction

ConTroll[1] developed out of the Troll system (Gerdemann et al., 1995). Troll is a phrase structure backbone system very similar to ALE (Carpenter and Penn, 1994), but it differs from that system in that it is based on the set theoretic logic of (King, 1989, 1994) rather than the information theoretic one of (Carpenter, 1992). The practical effect of this is that Troll implements an exhaustive typing strategy which provides the stronger kind of inferencing over descriptions (Gerdemann and King, 1993, 1994) required by standard HPSG theories.

We begin with a brief overview of the ConTroll architecture as shown in Fig. 1 before focusing on the aspects relevant to large scale grammar development. ConTroll supports two basic kinds of grammar constraints: universal implicational constraints with complex antecedents, and definite relations. As an example of the first kind of constraint, consider the Head Feature Principle of HPSG (Pollard and Sag, 1994). This universal constraint can be directly encoded in ConTroll as follows:

```
phrase, dtrs:headed_struc ==>
              synsem:loc:cat:head:  X,
              head_dtr:synsem:loc:cat:head:  X.
```

The ConTroll system allows a direct implementation of HPSG grammars without forcing the grammar writer to introduce a phrase structure backbone or recode the theories as logic programs. In addition, the availability of universal constraints in ConTroll also allows for a more modular encoding of traditional grammars using a relational backbone. This is so since in a relational encoding all subcases need to be covered. The universal constraints with complex antecedents, on the other hand, generalize over all occurrences of some data structure and can attach the constraint to the relevant subset. Universal constraints are thus constraint-based in the intuitive sense: each structure which is not explicitly excluded is well-formed. Internally, the complex antecedents of such universal constraints and the occurrences of negation are eliminated, which is possible due to the exhaustive typing we assume. The resulting type constraints are then compiled into definite clauses using the method described in (Götz and Meurers, 1995).

The second kind of grammar constraints are ordinary definite clauses with feature term arguments. The compiler detects places in which constrained types can occur and integrates the type constraints into the code by adding calls to the relational encoding of the universal constraints. As described in (Götz and Meurers, 1996), the universal constraints are integrated in a lazy fashion, i.e. only in case the argument of a relation is specific enough to cause a conflict with a universal constraint does the compiler attach a call to the universal constraint. Such

[*]The authors are listed alphabetically.

[1]ConTroll was developed in the B4 project of the SFB 340, funded by the Deutsche Forschungsgemeinschaft (DFG). The following people contributed to the development of the system: Dale Gerdemann and Erhard Hinrichs (project leaders), Björn Aldag, Natali Alt, Carsten Hess, John Griffith, Stephan Kepser, Guido Minnen, Gerald Penn, Oliver Suhre and Andreas Zahnert.

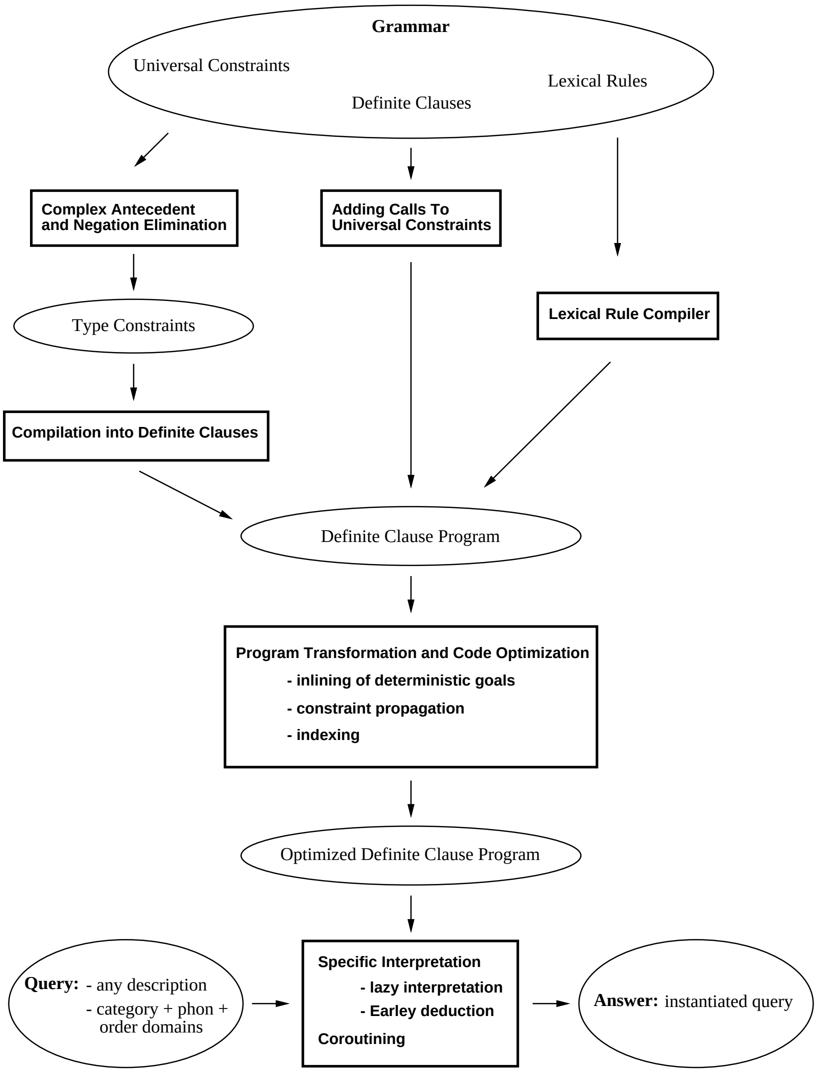

Figure 1: Overview of the ConTroll system

lazy interpretation has a significant efficiency payoff, especially for large grammars, since it results in preferred processing of those constraints in the large constraint pool which are specific enough to fail.

Special mechanisms are included to allow the grammar writer to specify how the universal constraints and definite clauses are intended to interleave in processing (Götz and Meurers, 1997). Most importantly, the delay mechanism of ConTroll supports coroutining of both universal constraints and definite clauses, and it offers a variety of control statements to fine tune the execution strategy. This is a prerequisite to efficient processing with constraint grammars.

For the rest of the paper, we will focus on those aspects of ConTroll, which directly address large scale grammar development:

- A graphical user interface:
  - data structure visualization
  - debugging and tracing tool
- Grammar organization issues:
  - supporting a modular file organization
  - automatic macro detection
  - automatic macro generation
- Compilation techniques:
  - special compilation of lexica with lexical rules for a compact and efficient lexicon
  - incremental compilation and global grammar optimization
  - arbitrary multiple indexing of constraints

ConTroll is the only system combining all of these features in one architecture. TFS (Emele and Zajac, 1990) also offered type constraints and relations and to our knowledge was the first working typed feature systems. However, it had some serious drawbacks. TFS did not allow universal constraints with complex antecedents, but only type constraints. And the system did not include a delay mechanism, so that it was often impossible to ensure termination or efficient processing.

ALE (Carpenter and Penn, 1994) provides relations and type constraints (i.e., only types as antecedents), but their unfolding is neither lazy, nor can it be controlled by the user in any way. This can lead to severe termination problems with recursive constraints. However, the ALE type constraints were designed to enhance the typing system, and not for recursive computation.

The definite clause part of our system is very similar to the one of CUF (Dörre and Dorna, 1993): both use delay statements and preferred execution of deterministic goals. CUF, however, does not offer universal constraints.

2 A graphical user interface

Two practical problems arise once one tries to implement larger grammars. On the one hand, the complex data structures of such grammars contain an overwhelming number of specifications which are difficult to present to the user. On the other hand, the interaction of grammar constraints tends to get very complex for realistic linguistic theories.

2.1 Data Structure Visualization

In ConTroll, the powerful graphical user interface Xtroll addresses the presentation problem. The Xtroll GUI programmed by Carsten Hess allows the user to interactively view AVMs, search attributes or values in those representations, compare two representations (e.g. multiple results to a query) and highlight the differences, etc. Fonts and Colors can be freely assigned to the attributes and types. The displayed structures (or any part of it) can be can be printed or saved as postscript file. The GUI comes with a clean backend interface and has already been used as frontend for other natural language applications, e.g., in the VERBMOBIL project.

A special feature of Xtroll is that it offers a mechanism for displaying feature structures as trees according to user specified patterns. Note that displaying trees is not an obvious display routine in ConTroll, since the system does not impose a phrase structure backbone but rather allows a direct implementation of HPSG grammars which usually encode the constituent structure under DTRS or some similar attribute. Since trees are a very compact representation allowing a good overview of the structure, Xtroll allows the user to specify that certain paths under a type are supposed to be displayed in a tree structure. As labels for the tree nodes, Xtroll can display a user definable selection of the following: the feature path to the node, the type of the structure, the phonology, and finally an abbreviation resulting from matching user specified feature structure patterns. An example for such a tree output is shown in Fig. 2. In this tree, the abbreviations were used to display category information in an X-bar fashion. Clicking on the labels displays the AVM associated with this node. In the example, we did open three of the nodes to show the modification going on between the adjective *schnelles (fast)* and the noun *fahrrad (bike)*. Note that those attributes which are irrelevant to this matter were hidden by clicking on those attributes.

The use of the fully user definable, sophisticated display possibilities of Xtroll in our experience have turned out to be indispensable for developing large typed feature based grammars.

2.2 A graphical debugger

The second problem is addressed with a sophisticated tracing and debugging tool which was developed to allow stepwise inspection of the complex constraint resolution process.

The debugger displays the feature structure(s) to be checked for grammaticality and marks the nodes on which constraints still have to be checked. As a result of the determinacy check, each such node can also be marked as failed, delayed or deterministic. Similar to standard Prolog debuggers, the user can step, skip, or fail a constraint on a node, or request all deterministic processing to be undertaken. An interesting additional possibility for non-deterministic goals is that the user can inspect the matching defining clauses and chose which one the system should try.

For example, in Fig. 3, the selected goal with tag $\boxed{1}$ is listed as delayed and is displayed at the bottom to have two matching defining clauses out of seven possible ones. Using the mouse, the user can chose to display the matching or all defining clauses in separate windows.

We believe that the availability of a sophisticated debugger like the one implemented for the ConTroll system is an important prerequisite for large scale grammar development.

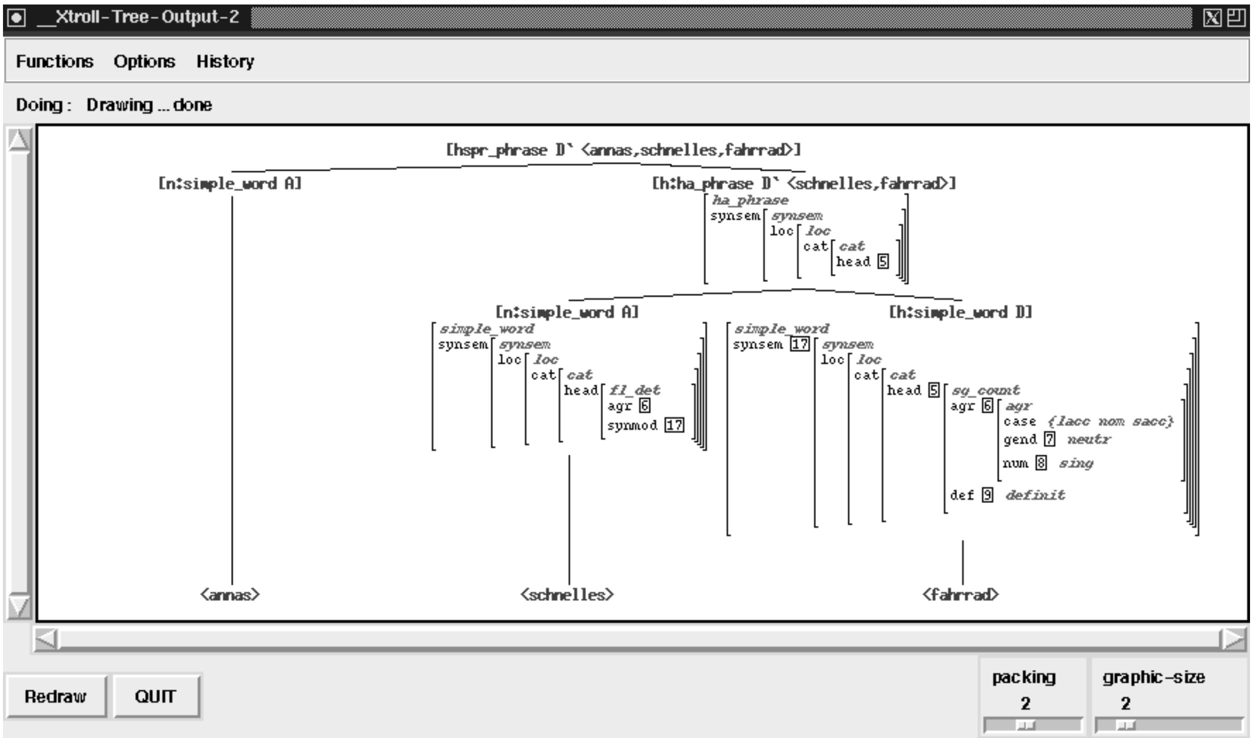

Figure 2: Screen shot of a window showing user configured tree output

3 Grammar Organization Issues

3.1 A modular grammar file organization

To organize grammars in a modular fashion, it is important to be able to distribute the grammar into several files to permit modification, loading, and testing of the different parts of a grammar separately. Also, only a modular file organization allows for distributed grammar development since such an organization makes it possible to coordinate the work on the individual files with software engineering tools such as a revision control system.

ConTroll supports the use of a grammar configuration file which can contain basic directory and file specifications, as well as a more sophisticated system allowing the linguist to specify the dependencies between signature, theory and lexicon files.

To find out which signature, theory, and lexicon is supposed to be used and in which directories the files are supposed to be found, the system looks for a grammar configuration file. If such a file is not found, default names for the signature and the theory file are used. If there is a configuration file, it can specify the theory, signature, and lexicon files to be used, as well as the relevant directories. The downside of this explicit mode of specification is that each time one wants to load a part of the grammar, e.g. for testing, one needs to realize which files needed are needed to test this part of the grammar in order to be able to list them explicitly. While this might seem like a trivial task, our experience has shown that in a distributed grammar development environment such a complete specification requires significant insight into the entire grammar.

ConTroll therefore provides a more intelligent way of specifying the files needed for a module by allowing statements which make the dependencies between the different theory, signature, and lexicon files explicit. These specifications were modeled after the makefiles used in some programming environments. Once the dependencies are provided in the configuration file, selecting the parts of the grammar to be loaded can be done without having to being aware of the whole grammar organization by specifying one file for each module of the grammar needs to be included. The signature, theory, and lexicon files which are needed for the selected files are then automatically loaded according to the dependency specifications.

3.2 Automatic macro detection

When writing a typed feature structure based grammar one usually wants to abbreviate often used feature paths or complex specifications. In ConTroll this can be done using the definite clause mechanism. However, from a processing point of view, it is inefficient to treat macros in the same way as ordinary relations. We thus implement a fast, purely syntactic preprocessing step that finds the relations that can be treated as macros, i.e., unfolded at compile time. These macro relations are then compiled into an internal representation during a second preprocessing step. When the rest of the grammar is parsed, any macro goal will simply be replaced by its internal representation.

After the actual compilation, ConTroll closes the grammar under deterministic computation. This step must be carefully distinguished from the macro detection described above. A *goal* is deterministic in case it matches at most one defining clause, but a *relation* is a macro by virtue of its definition, irrespective of the instantiation of the actual calling goals. Of course, the macro detection step can be eliminated, since the deterministic closure will also unfold all macros. However, for our reference grammar, adding the macro detection step reduced compile times by a factor of 20. Thus, for large grammars, compilation without macros is simply not practical.

Obviously, making automatic macro detection a property of the compiler relieves the grammar developer from the burden of distinguishing between macros and relations, thereby eliminating a potential source of errors.

3.3 Automatic macro generation

Since HPSG theories usually formulate constraints about different kind of objects, the grammar writer usually has to write a large number of macros to access the same attribute, or to make the same specification, namely one for each type of object which this macro is to apply to. For example, when formulating immediate dominance schemata, one wants to access the VFORM specification of a *sign*. When specifying the valence information one wants to access the VFORM specification of a *synsem* object. And when specifying something about non-local dependencies, one may want to refer to VFORM specifications of *local* objects.

ConTroll provides a mechanism which automatically derives definitions of relations describing one type of object on the basis of relations describing another type of object – as long as the linguist tells the system which path of attributes leads from the first type of object to the second.

Say we want to have abbreviations to access the VFORM of a *sign*, a *synsem*, *local*, *cat*, and a *head* object. Then we need to define a relation accessing the most basic object having a VFORM, namely *head*: `vform_h(X-vform) :== vform:X.` Second, (once per grammar) `access_suffix` and `access_rule` declarations for the grammar need to be provided. The former define a naming convention for the generated relations by pairing types with relation name suffixes. The latter define the rules to be used by the mechanism by specifying the relevant paths from one type of object to another. For our example the grammar should include the recipes shown in Fig. 4. This results in the macros shown in Fig. 5 to be generated.

```
access_suffix(head,"_h").
access_suffix(cat,"_c").
access_suffix(loc,"_l").
access_suffix(synsem,"_s").
access_suffix(sign,"someSuffix").

access_rule(cat,head,head).
access_rule(loc,cat,cat).
access_rule(synsem,loc,loc).
access_rule(sign,synsem,synsem).
```

Figure 4: Macro generation specification

```
vform_h(X)   :=   vform:X.
vform_c(X)   :=   head:vform_h(X).
vform_l(X)   :=   cat:vform_c(X).
vform_y(X)   :=   loc:vform_l(X).
vform_s(X)   :=   synsem:vform_y(X).
```

Figure 5: Example result of macro generation

For a large grammar, which usually specifies hundreds of macros, this mechanism can save a significant amount of work. It also provides a systematic rather than eclectic way of specifying abbreviations in a grammar, which is vital if several people are involved in grammar development.

4 Compilation techniques for large scale grammars

4.1 Lexical rules for a compact and efficient lexicon encoding

Lexical rules receive a special treatment in ConTroll. The lexical rule compiler implements the covariation approach to lexical rules (Meurers and Minnen, 1995). It translates a set of HPSG lexical rules and their interaction into definite relations used to constrain lexical entries. In HPSG, lexical rules are intended to "preserve all properties of the input not mentioned in the rule." (Pollard and Sag, 1987, p. 314). The lexical rule compiler of the ConTroll system to our knowledge is the only system which provides a computational mechanism for such lexical rules by automatically computing the necessary frame predicates accounting for the intended preservation of properties. Since the lexical rules do not need to be expanded at compile time, ConTroll is able to handle the infinite lexica which have been proposed in a number of HPSG theories.

Constraint propagation is used as program transformation techniques on the definite clause encoding resulting from the lexical rule compiler (Meurers and Minnen, 1996). The relation between parsing times with the expanded (EXP), the covariation (COV) and the constraint propagated covariation (OPT) lexicon for a German HPSG grammar (Hinrichs, Meurers, and Nakazawa, 1994) can be represented as OPT : EXP : COV = 0.75 : 1 : 18. Thus, the lexical rule compiler results not only in a compact representation but also in more efficient processing of a lexicon including lexical rules.

4.2 Incremental compilation and global grammar optimization

To keep development cycles short, a fast compiler is essential. Particularly when developing a large grammar, small changes should not necessitate the recompilation of the whole grammar – an incremental compiler is called for. This is relatively easy for systems where the compilation of individual pieces of code does not depend on the rest of the program. In ConTroll, this task is complicated for two reasons.

1. **Interaction of universal constraints.** If several different universal constraints apply to objects of the same type, the compiler will merge them together. Changing a single high-level constraint may thus necessitate the recompilation of large parts of the grammar.

2. **Off-line deterministic closure.** Since the grammar is closed under deterministic computation at compile time, a change in some relation entails recompilation of all clauses that have inlined a call to that relation, which in turn may lead to changes in yet other relations, and so on. Nothing less than the maintenance of a complete call graph for the whole grammar would enable the compiler to know which parts of the grammar need to be recompiled.

We decided on a compromise for incremental compilation and made our compiler aware of the first sort of dependency, but not the second. This means that incremental recompilation is always done on the basis of the grammar before deterministic closure. Therefore, after incremental recompilation deterministic closure needs to be done for the whole grammar.

4.3 Arbitrary multiple indexing of grammar constraints

ConTroll allows the specification of indexing information for predicates individually. This is comparable to the indexing of terms in relational databases, e.g., the SICStus Prolog external database (Nilsson, 1995). Figure 6 shows the definition of a two-place

```
r(t)   **>   t.
r(a)   :=    b.    index(r,arg0:t).
r(a)   :=    c.    index(r,arg1:t).
r(b)   :=    c.
```

Figure 6: Indexing specification for `r`

relation `r` including a typing declaration and two indexing instructions. Given a fully instantiated goal for the relation `r`, the run-time environment of ConTroll can deterministically pick the right clause without leaving behind a choice-point.

The indexing mechanism not only works for relations, but also implicational constraints. Figure 7 shows possible indexing instructions for the lexical

```
index(word,phon:hd:string).
index(word,synsem:loc:cat:head:head).
```

Figure 7: Indexing for the type `word`

type `word`, namely for the phonological form, and the syntactic category.

5 Experience using the System

Our implementation has been tested with several smaller and one large (> 5000 lines) grammar, a linearization-based grammar of a sizeable fragment of German. The grammar was developed in a distributed fashion by eight people and consist of 57 files. It provides an analysis for simple and complex verb-second, verb-first and verb-last sentences with scrambling in the Mittelfeld, extraposition phenomena, *wh*-movement and topicalization, integrated verb-first parentheticals, and an interface to an illocution theory, as well as the three kinds of infinitive constructions (coherent, incoherent, third-construction), nominal phrases, and adverbials (Hinrichs et al., 1997).

With grammars this size, it is necessary to pay careful attention to control to achieve acceptable parsing times. With our Prolog based interpreter, parse times were around 1-5 sec. for 5 word sentences and 10-60 sec. for 12 word sentences. We are currently experimenting with a C based compiler (Zahnert, 1997) using an abstract machine with a specialized set of instructions based on the WAM

(Warren, 1983; Aï-Kaci, 1991). This compiler is still under development, but it is reasonable to expect speed improvements of an order of magnitude.

6 Summing Up

We characterized ConTroll as a system supporting grammars expressed with definite relations, implicational constraints with complex antecedents, and lexical rules. To achieve an efficient interpretation, implicational constraints are applied in a lazy fashion, and interleaving of execution of the different constraints can be determined by the grammar writer using delays and other directives.

We focussed on those properties of ConTroll which we take to be indispensable for large scale grammar development: a graphical user interface for data structure visualization, a sophisticated debugging and tracing tool, support for a modular file organization, a special macro treatment, and finally a set of special compilation techniques such as an incremental compiler, a mechanism dealing with lexical rules in an efficient way, and a way to use multiple indexing of grammar constraints for efficient access to large constraint pools.

References

Aï-Kaci, Hassan. 1991. *Warren's Abstract Machine.* MIT Press.

Carpenter, Bob. 1992. *The logic of typed feature structures*, volume 32 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press.

Carpenter, Bob and Gerald Penn. 1994. ALE – The Attribute Logic Engine, User's Guide, Version 2.0.1, December 1994. Technical report, Carnegie Mellon University.

Dörre, Jochen and Michael Dorna. 1993. CUF - a formalism for linguistic knowledge representation. In Jochen Dörre, editor, *Computational aspects of constraint based linguistic descriptions I.* DYANA-2 Deliverable R1.2.A, Universität Stuttgart, August, pages 1–22.

Emele, Martin C. and Rémi Zajac. 1990. Typed unification grammars. In *Proceedings of the* 13^{th} *International Conference on Computational Linguistics.*

Gerdemann, Dale, Thilo Götz, John Griffith, Stephan Kepser, and Frank Morawietz, 1995. *Troll manual.* Seminar für Sprachwissenschaft, Universität Tübingen, draft edition, October.

Gerdemann, Dale and Paul John King. 1993. Typed feature structures for expressing and computationally implementing feature cooccurence restrictions. In *Proceedings of 4. Fachtagung der Sektion Computerlinguistik der Deutschen Gesellschaft für Sprachwissenschaft*, pages 33–39.

Gerdemann, Dale and Paul John King. 1994. The correct and efficient implementation of appropriateness specifications for typed feature structures. In *Proceedings of COLING-94*, Kyoto, Japan.

Götz, Thilo and Walt Detmar Meurers. 1995. Compiling HPSG type constraints into definite clause programs. In *Proceedings of the Thrirty-Third Annual Meeting of the ACL*, Boston. Association for Computational Linguistics.

Götz, Thilo and Walt Detmar Meurers. 1996. The importance of being lazy - using lazy evaluation to process queries to HPSG grammars. In Philipe Blache, editor, *Actes de la troisième conférence anuelle sur le traitment automatique du langage naturel.*

Götz, Thilo and Walt Detmar Meurers. 1997. Interleaving universal principles and relational constraints over typed feature logic. In *Proceedings of the 35th Annual Meeting of the ACL and the 8th Conference of the EACL*, Madrid, Spain.

Hinrichs, Erhard, Detmar Meurers, and Tsuneko Nakazawa. 1994. Partial-VP and Split-NP topicalization in German — An HPSG analysis and its implementation. Arbeitspapiere des SFB 340 Nr. 58, Universität Tübingen.

Hinrichs, Erhard, Detmar Meurers, Frank Richter, Manfred Sailer, and Heike Winhart. 1997. Ein HPSG-Fragment des Deutschen, Teil 1: Theorie. Arbeitspapiere des SFB 340 Nr. 95, Universität Tübingen.

King, Paul John. 1989. *A logical formalism for head-driven phrase structure grammar.* Ph.D. thesis, University of Manchester.

King, Paul John. 1994. An expanded logical formalism for head-driven phrase structure grammar. Arbeitspapiere des SFB 340 Nr. 59, Universität Tübingen.

Meurers, Walt Detmar and Guido Minnen. 1995. A computational treatment of HPSG lexical rules as covariation in lexical entries. In *Proceedings of the Fifth International Workshop on Natural Language Understanding and Logic Programming*, Lisbon, Portugal.

Meurers, Walt Detmar and Guido Minnen. 1996. Off-line constraint propagation for efficient HPSG processing. In *HPSG/TALN Proceedings*, Marseille, France.

Nilsson, Hans. 1995. The external storage facility in SICStus Prolog. Technical report R91:13, Swedish Institute of Computer Science.

Pollard, Carl and Ivan A. Sag. 1987. *Information-based Syntax and Semantics, Vol. 1*. Number 13 in Lecture Notes. CSLI Publications, Stanford University. Distributed by University of Chicago Press.

Pollard, Carl and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar*. University of Chicago Press, Chicago.

Warren, David H. D. 1983. An abstract Prolog instruction set. Technical note 309, SRI International.

Zahnert, Andreas. 1997. fl2c – ein Compiler für CLP(TFS). Diplomarbeit, Fakultät für Informatik, Universität Tübingen.

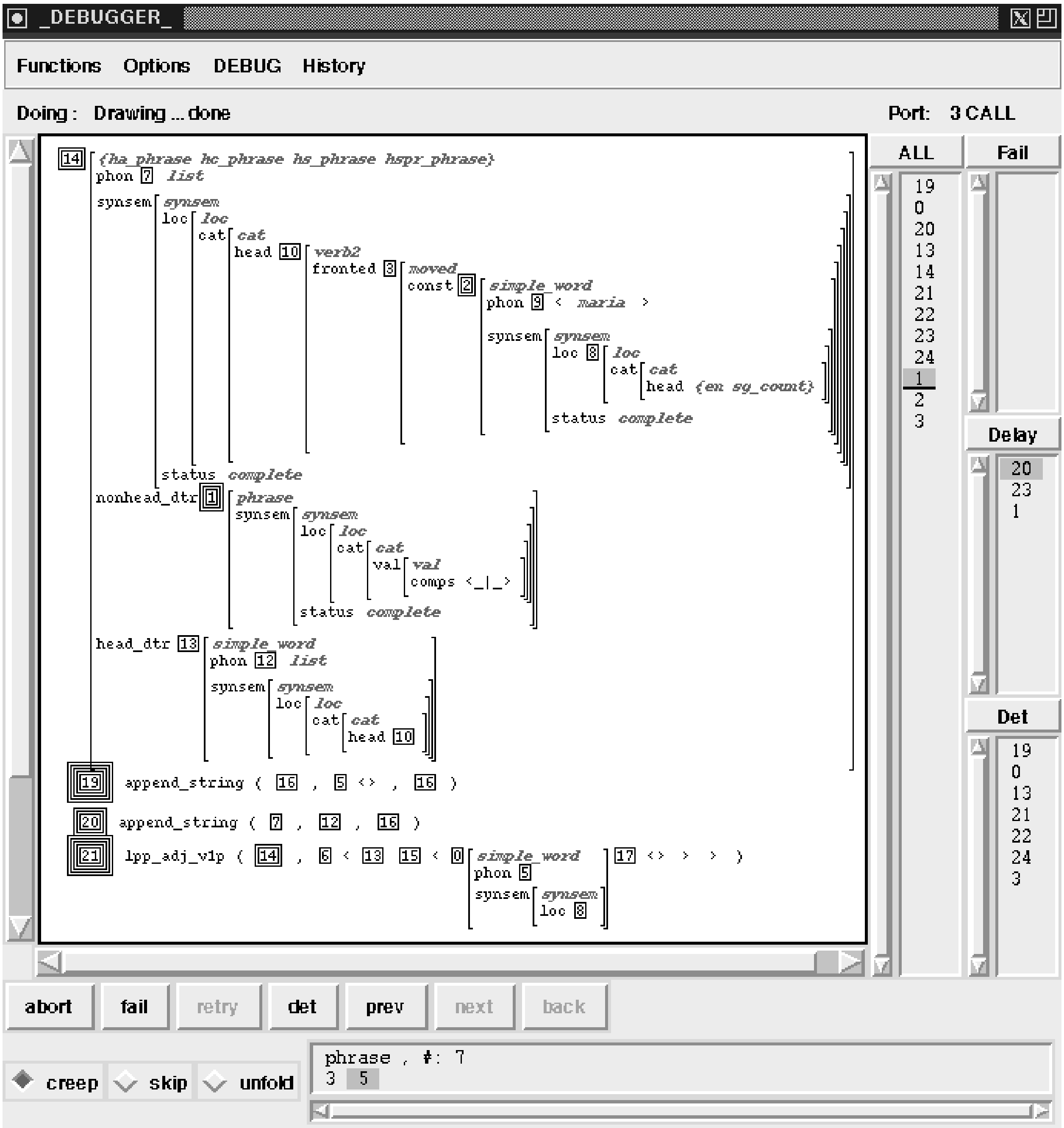

Figure 3: Screen shot of the graphical debugger in action