Kahina: A Hybrid Trace-Based and Chart-Based Debugging System for Grammar Engineering

Johannes Dellert[1], Kilian Evang[2], and Frank Richter[1]

[1] University of Tübingen
[2] University of Groningen

Abstract. This paper provides an overview of the debugging framework Kahina, discussing its architecture as well as its application to debugging in different constraint-based grammar engineering environments. The exposition focuses on and motivates the hybrid nature of the system between source-level debugging by means of a tracer and high-level analysis by means of graphical tools.

1 Introduction

Several decades after their inception, the design, implementation and debugging of symbolic, constraint-based grammars is still a considerable challenge. Despite all efforts to modularize grammars, declarative constraints can have far-reaching unexpected side effects at runtime. The original grammar writers are often linguists with little experience in software development, meaning that in practice their declarative designs must be optimized for the algorithmic environment by iterative refinement. It is a major challenge in this scenario that the execution of even relatively simple grammars tends to result in many thousand computation steps. To make debugging of such grammars feasible at all, innovative and carefully designed debugging tools are needed.

In this paper, we present central aspects of the Kahina debugging framework, which was created to address these issues. We discuss its general architecture as well as its application to grammar debugging in two implementation platforms for constraint-based grammars, TRALE [3] and QType [15]. Unlike other grammar engineering environments, Kahina emphasizes the analysis of the execution model of declaratively designed grammars, with the goal of making the procedural effects of constraint systems more transparent. Developers can spot more easily where constraints have adverse side effects, and where an implementation can be optimized for efficiency. The main challenge in creating such a debugging system is to find appropriate ways to project functionally meaningful intervals from large sequences of computations onto intuitively comprehensible graphical displays. Within the overall display, the close connections between the different perceptual units need to be highlighted, while at the same time avoiding to produce redundant information that could easily overwhelm the user. A highly configurable selection of various perspectives on grammar execution is offered to make its contingencies explorable in a balanced fashion.

Kahina was initially developed as a graphical front end for TRALE's source-level debugger. TRALE is an interesting target, as it is explicitly designed for the implementation of grammatical constraints that stay close to the specification of theoretical HPSG grammars. As Melnik [13] finds in her comparison of the LKB [4] and TRALE for HPSG grammar implementation, the latter system requires fewer initial adjustments of theoretical grammars, at the price of a possibly stronger deviation of the processing model from non-technical user expectations. At the same time, the LKB traditionally provided more graphical debugging support and guidance to users, similar to the facilities featured by XLE [11], an implementation environment for LFG. Kahina aims at graphical debugging support for complex implementation systems such as TRALE, especially to help novice linguist users understand the underlying procedural model. It is designed to bridge the considerable gap between detailed but low-level, command-line based debugging for expert users, and the high-level view of chart-based tools, which (deliberately) hide many potentially relevant procedural details.

Beyond support for grammar implementation platforms, Kahina provides advanced general debugging facilities for logic programming. The system expands on earlier ideas for graphical Prolog debugging presented by e.g. Dewar & Cleary [8] and Eisenstadt [9], and is also inspired by SWI-Prolog's GUI tracer [16], the most mature visualization tool for Prolog processes currently available.[3]

Section 2 gives a bird's-eye view of the Kahina architecture, also outlining the process of implementing a debugging system. Sections 3–5 focus on the central features and functions of a debugging system for TRALE. Section 6 discusses chart displays as a central component of many grammar development systems, and provides a case study of how demands of different systems are accommodated. Section 7 critically evaluates Kahina's approach to grammar debugging, before Section 8 summarizes prominent features and points to open issues.

2 The Kahina Architecture

Kahina is written in Java and distributed under the GPL.[4] Core components are a GUI framework based on Swing, a State class for storing large amounts of step data, a message passing system (Controller) for communication among components, and a control agent system for automatization of user interactions.

Kahina is designed as a general framework for implementing debugging systems, by which we mean integrated graphical environments for analyzing computations consisting of hierarchically related steps, where each step may be associated with a source code location and other detail information for visual display. A debugging system is built by implementing specialized components using Kahina's classes and interfaces. The architecture of the TRALE debugger is shown in Fig. 1; a similar architecture has been used to implement graphical debuggers for different client systems, including SICStus and SWI Prolog.

[3] An earlier version of Kahina for TRALE was previously presented in a poster session at the HPSG conference 2010 [7].

[4] http://www.kahina.org

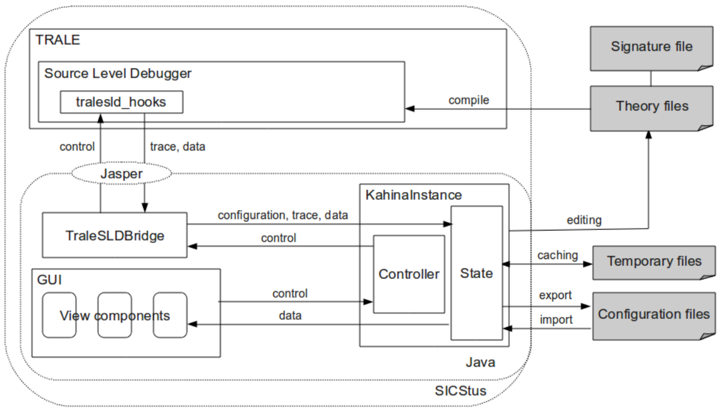


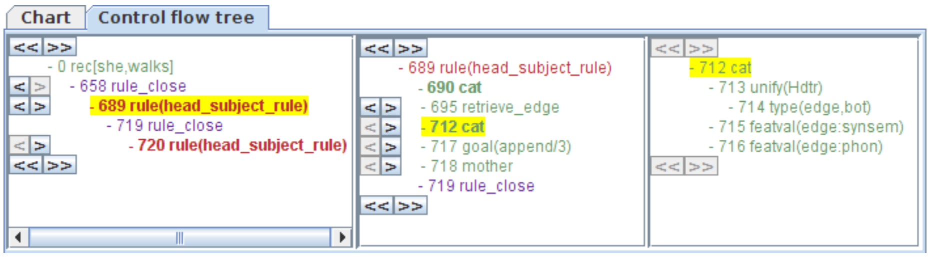
Fig. 1: Architecture of the Kahina-based TRALE debugger

On the Prolog side, the existing source-level debugger was extended by a control loop that interleaves with TRALE's execution process and communicates with Kahina to transmit step details and obtain tracing commands for controlling execution. Communication with Kahina is done via SICStus Prolog's Jasper library, which uses the Java Native Interface. On the Java side it is handled by a specialized **bridge** which translates low-level TRALE tracing information into Kahina's data model for storage and visualization. Another important application-specific Java component is a **step** class used by the State component, defining the data attributes associated with each step. Finally, a customizable configuration of **view components** defines how the step data are visualized.

3 Visualizing Parsing Processes

In this and the next two sections, we focus on the TRALE debugger and the features it offers to grammar engineers. In TRALE, a chart parser steps through the input from right to left, building all possible chart edges starting at each position before proceeding. Its principal operations or *steps* are:

- `rule_close`: A failure-driven loop over all phrase-structure rules in the grammar, takes a chart edge as input and recursively builds all edges that can be built with the selected edge as the leftmost child.
- `rule`: Tries to apply a phrase-structure rule with the input edge as leftmost daughter. Existing chart edges are used for covering the other daughters. Success leads to a new edge on which `rule_close` is called recursively.
- `retrieve_edge`: Retrieves a passive edge from the chart for use as a non-leftmost daughter in a `rule` application.
- `cat`: Applies a daughter description as part of a `rule` application.
- `goal`: Executes a procedural attachment as part of a `rule` application.
- `mother`: Applies a mother description as part of a `rule` application.



```
- 0 rec[she,walks]
  - 658 rule_close
    - 689 rule(head_subject_rule)
      - 719 rule_close
        - 720 rule(head_subject_rule)
```

```
- 689 rule(head_subject_rule)
  - 690 cat
  - 695 retrieve_edge
  - 712 cat
  - 717 goal(append/3)
  - 718 mother
  - 719 rule_close
```

```
- 712 cat
  - 713 unify(Hdtr)
    - 714 type(edge,bot)
  - 715 featval(edge:synsem)
  - 716 featval(edge:phon)
```

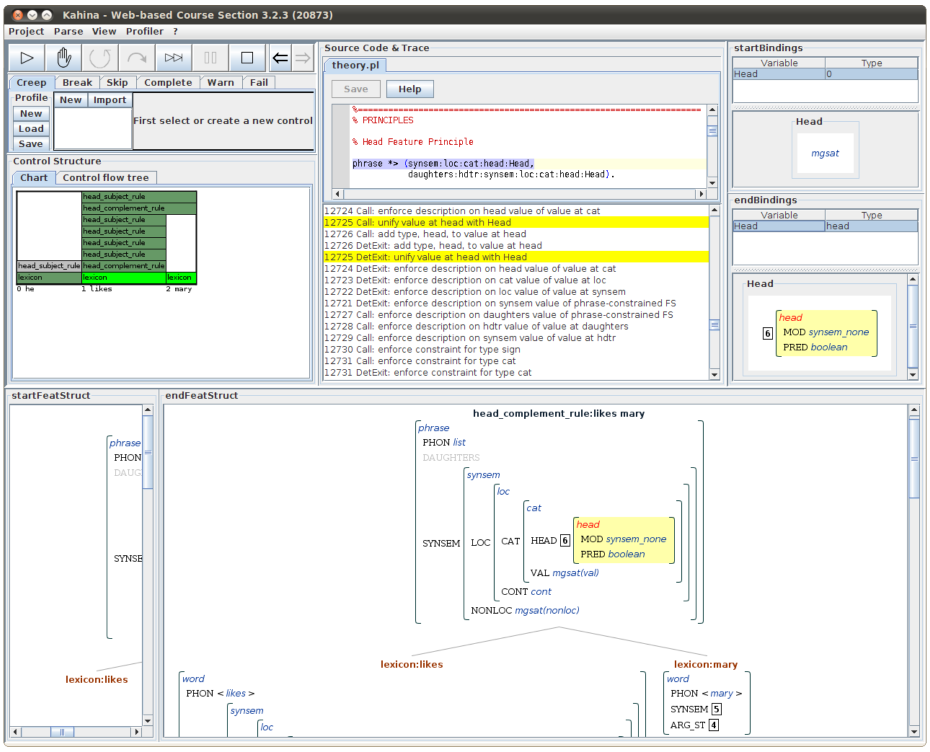
Fig. 2: Control flow tree of a parsing process with three levels of detail

TRALE comes with a command-line based tracer which treats these steps as *procedure boxes* following the Prolog tracing model of [1]. Kahina builds upon this notion and visualizes parsing processes as trees with goals as nodes and subgoals as child nodes. The resulting **control flow tree** is a central element of Kahina's GUI and is used to retrieve details on individual steps (see Section 4) by mouseclick. Each node is labeled with a numeric ID and a short description of the step, and is color-coded for status (Call, Exit, DetExit, Redo or Fail).

Since a single huge tree with thousands of parse steps cannot be navigated, the tree is split into three subviews that show different levels of detail (Fig. 2). The first subview shows a thinned-out version of the tree in which only the **cornerstones** are displayed, i.e. the `rule_close` and `rule` steps. Selection of cornerstone nodes controls which part of the tree occupies the second subview: it contains the descendants of the selected cornerstone down to the next cornerstones, displayed as leaves. Descendants of `cat`, `goal` and `mother` nodes are in turn displayed in the third subview when the respective step is selected.

Apart from their size, TRALE parsing processes are challenging to visualize due to their complex structure. Steps can be arranged in at least two meaningful tree structures. The **call tree**, in which goals have their subgoals as children, is visualized in the control flow tree through indentation. This tree roughly corresponds to the structure of clauses and subclauses in a logic program. The **search tree** is used in backtracking. Without backtracking, a search tree would be a long unary branch in which each step is the child of the step invoked before it, visualized by the top-to-bottom arrangement of steps in the control flow tree. When Prolog backtracks, the step that is the last active choicepoint is copied to represent the new invocation. The copy becomes a sibling of the original, starting a new branch. Kahina shows only one branch of the search tree at a time, focusing on visualizing the call tree, but at each choicepoint two arrow buttons permit browsing through siblings in the search tree. Access to earlier, "failed" branches is important because TRALE makes extensive use of failure-driven loops for exhausting the search space of possible edges. The user can flip through the different rules that were applied to any particular edge.

One way to inspect parsing processes is to trace them interactively, watching the tree grow step by step or in larger chunks. The available tracing commands are similar to those of a classical Prolog tracer (e.g. SICStus Prolog [2]). They



```
%==========================================================================
% PRINCIPLES

% Head Feature Principle

phrase *> (synsem:loc:cat:head:Head,
           daughters:hdtr:synsem:loc:cat:head:Head).
```

```
12724 Call: enforce description on head value of value at cat
12725 Call: unify value at head with Head
12726 Call: add type, head, to value at head
12726 DetExit: add type, head, to value at head
12725 DetExit: unify value at head with Head
12724 DetExit: enforce description on head value of value at cat
12723 DetExit: enforce description on cat value of value at loc
12722 DetExit: enforce description on loc value of value at synsem
12721 DetExit: enforce description on synsem value of phrase-constrained FS
12727 Call: enforce description on daughters value of phrase-constrained FS
12728 Call: enforce description on hdtr value of value at daughters
12729 Call: enforce description on synsem value of value at hdtr
12730 Call: enforce constraint for type sign
12731 Call: enforce constraint for type cat
12731 DetExit: enforce constraint for type cat
```

Fig. 3: Kahina's GUI showing tracing buttons, the chart, and all step detail views

are exposed as clickable buttons, shown in Fig. 3: `creep` advances to the next step. `fail` forces a not-yet-completed step to fail, making it possible to explore otherwise inaccessible parts of the search space. `auto-complete` completes the current step without interaction at substeps, but saves all information about the substeps for later inspection. `skip` does the same without collecting any tracing information. In effect it prunes parts of the process currently not of interest. `leap` proceeds without interaction up to the next breakpoint (see Section 5) or the end of the parsing process. Additional buttons pause or stop leap and auto-complete actions, or step through a history of recently selected steps. Since Kahina keeps old backtracking branches available – a feature that sets it apart from other graphical Prolog tracers such as that of SWI-Prolog [16] – it also supports full post-mortem analysis of a completed parsing process.

The **message console** complements the control flow tree as a navigation tool by displaying a timeline of tracing events. Events include step ports just like in a console-based Prolog tracer, but also user-generated and automatized tracing events: forced fails, auto-completes, skips, and leaps. All events associated with the currently selected step are highlighted, such as the Call port and the DetExit port of the selected `unify` step in Fig. 3.

4 Step Detail Views

Kahina's GUI contains different views for displaying details of the currently selected step. Steps are selected dynamically according to the context, in most cases through direct user interaction with either the control flow tree or the **chart display**, a graphical representation of the parse chart (Section 6).

The **source code editor** reacts to the selection of a step that corresponds to an element in the grammar (a rule application, a description, a constraint or a procedure) by marking the corresponding source code line. Computation steps are thus connected to the underlying grammar code. Fig. 3 shows the step details of a unification step within a constraint application (HPSG's Head Feature Principle) that fires during the application of a description of a rule to the mother feature structure. The defining source code line is colored. Syntax highlighting and basic editing are also supported.

While the source code view shows the reasons for what is happening, the **feature structure view** shows the structures that are being built as a result. For this view, Kahina draws on the Gralej[5] library. For steps that belong to rule applications, it shows the corresponding local tree, with feature structures for all daughters that have been or are being processed, and also the mother structure once it is being processed. The substructure that is being modified in each step is highlighted in yellow. For steps that belong to procedural attachments, the active goal is displayed with embedded graphical representations for feature structure arguments. A separate **bindings view** analogously shows feature structures associated with *variables* in the TRALE description language at every point during the application of a rule, constraint, or procedure.

In Fig. 3, the head complement rule is being applied to the input substring "likes mary", as shown by the local tree in the feature structure view. The current constraint binds the mother's SYNSEM:LOC:CAT:HEAD to a variable. The effect is seen in the bindings view, which compares the state before and after execution of the step. Before, the Head variable is still unbound, defaulting to *mgsat* (most general satisfier). Afterwards, it contains the value of the head feature.

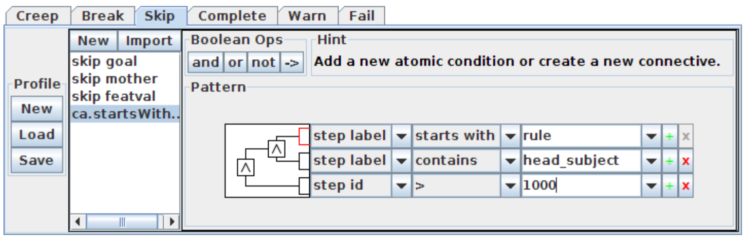
5 Automatization via Control Agents

Stepping through a parsing process manually and skipping or auto-completing irrelevant sections is error-prone and tiring. The `leap` command completes a parse without interaction for later post-mortem analysis, but may preserve too much information if only a certain part of the grammar should be debugged. Classic Prolog tracers, and also TRALE's console-based debugger, offer automatization via **leashing** and **breakpoints**. The user determines that certain types of steps should be **unleashed**, i.e. the tracer will simply proceed (`creep`) at them without asking what to do. Thus, the user can step through the parse faster, without having to issue a `creep` command at every step. Breakpoints single out a class

[5] http://code.google.com/p/gralej/

of steps of interest and then use the `leap` command to immediately jump to the next occurrence of that kind of step in the trace without further interaction.

Kahina generalizes leashing and breakpoints to the more powerful concept of **control agents**. A control agent is best seen as a simple autonomous agent consisting of a **sensor** which detects a pattern in individual steps or in a step tree, and an **actuator** which reacts to pattern matches by issuing tracing commands. Control agents are a method of describing and implementing intelligent behavior with the purpose of automatizing parts of the tracing and thereby of the grammar debugging process.



skip goal
skip mother
skip featval
ca.startsWith..

Add a new atomic condition or create a new connective.

step label	starts with	rule
step label	contains	head_subject
step id	>	1000

Fig. 4: The control agent editor

The control agent editor (Fig. 4) serves to define, activate and deactivate control agents. Control agents are grouped according to their tracing command, and are accordingly called creep agents, skip agents, etc. While creep agents typically effect unleashing behavior, skip agents and complete agents complete steps without interaction, including all descendant steps, making these steps atomic for the purposes of tracing. Break agents act like breakpoints by exiting leap mode when their sensors fire. Fail agents let certain steps fail immediately, which is useful e.g. for temporarily deactivating some phrase-structure rule. Warn agents have actuators which not only stop a leap but also display a warning. Their sensor counts the pattern matches, and only fires after reaching a predefined threshold. Warn points detect infinite recursion or inefficiencies, e.g. when a procedural attachment predicate is called too often.

Due to the flexibility in their actuators, control agents are more powerful than traditional debugging mechanisms. Since they build on the existing tracing commands rather than introducing completely new concepts[6], we believe they are more intuitive to learn after a short exposure to manual tracing. The central technique for defining basic control agents is the creation of sensors by defining **step patterns**. Usually a simple substring check suffices to identify the relevant step class, but users can also build complex conditions with logical connectives and a range of elementary step tests, as exemplified in Fig. 4. Additionally, **source code sensors** that fire at steps associated with a code location can be created by opening the source code view's context menu for the line of interest

[6] In fact, Kahina sees the manually-tracing user as just another, external control agent with especially complex behavior.

(such as the line containing the name of a phrase structure rule, in order to catch rule application steps) and selecting the desired actuator. In sum, control agents provide an expressive yet accessible tool for automatization, which is crucial for efficient tracing of parse processes.

6 Specialized Charts for TRALE and QType

Apart from storing partial results in parsing algorithms based on dynamic programming, a **chart** summarizes the parsing process by storing successfully parsed constituents, including those which do not become part of a complete parse. The spans covered by constituents are usually symbolized as **edges** over the input string. By default, Kahina's chart display shows currently **active edges** (gray) and successfully built **passive edges** (green). Dependencies between chart edges are optionally displayed by highlighting the edges the selected edge is composed of, and the edges that it is part of. An unexpected parse can often be narrowed down to an unwanted edge for a substructure, while a missing parse is often due to an unrecognized constituent. This has made chart displays a central top-level parse inspection tool in the LKB and XLE. Practical grammar engineering in TRALE has heavily relied on a third-party chart display. While parsing algorithms usually only require storage of positive intermediate results, a grammar engineer often needs to find out why an expected substructure is missing. Kahina recognizes the equal importance of successful and failed edges: A failed attempt to build an edge can be displayed as a **failed edge** which is linked to the step where the respective failure occurred. A chart that does not contain failed edges, such as the LKB chart, does not provide direct access to such information.

As the chart display is fully integrated with the basic tracing functionality, the number of edges on the chart grows as a parse progresses. Every chart edge constitutes a link into the step tree, giving quick access to the step where the respective attempt to establish a constituent failed or succeeded.

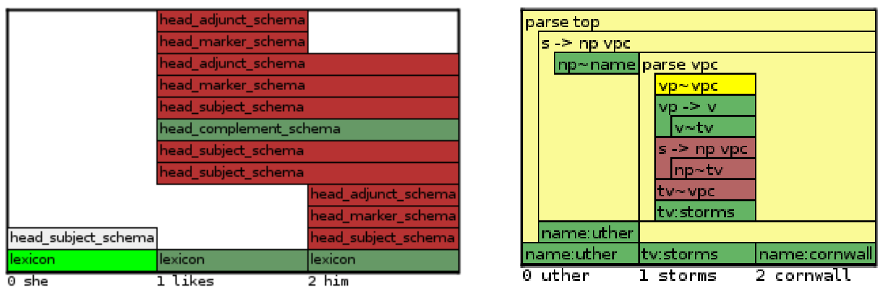


Fig. 5: Examples of the chart displays for TRALE and QType

Kahina's view components are designed for flexible customization to different grammar engineering systems, as demonstrated by two very different chart variants tailored to TRALE and QType, respectively. On the left side of Figure 5, we see a chart from a TRALE implementation of the English grammar by Pollard & Sag [14]. After recognizing all the input tokens, the right-to-left bottom-up parser has established an edge for the constituent “likes him” by means of the head complement schema. The chart also contains failed edges for several other schemas. The red edges below the successful edge represent failed attempts to form constituents by means of the respective lexicon edge. The edges above it represent failed attempts to build larger constituents from the successful edge. The lexicon edge for the input symbol “she” is currently being closed. The active application of the head subject schema will eventually lead to a successful parse.

Unlike TRALE, QType does not use dynamic programming. To still provide the functionality of a chart for analyzing left-corner parses, we defined a bridge that constructs a left-corner (LC) chart from the incoming step information. An intermediate stage of this chart for a parse process is displayed on the right side of Figure 5. The top-down prediction steps of the LC parser are visualized by inverted L-shaped prediction edges which wrap around the edges that were generated while attempting to complete the prediction. In the example, the active prediction edge labeled `s -> np vpc` ranging over the entire sentence indicates that QType is working on a parse based on the corresponding rule at the root level. The required `np` constituent has already been recognized by successful unification with the feature structure derived from the lexical entry for “uther”, and QType is now trying to complete the expected `vpc` constituent. The feature structure for the next token “storms” is of category `tv`. Attempts to interpret it directly as a `vpc`, or to once again apply the rule `s -> np vpc` to integrate it, have failed. The unary rule `vp -> v` was more successful, resulting in a feature structure of type `vp`. Its unification with the `vpc` specification in the rule definition will fail due to the mismatch between the subcat lists of the transitive verb and the postulated `vpc` structure, causing the correct phrase structure rule for verbs with one complement to be predicted next.

7 Discussion

One of the main contributions of Kahina to symbolic grammar engineering consists in the integration of two very different debugging paradigms of previous grammar debugging environments. We first compare Kahina to the old console-based source level debugger (SLD) for TRALE, and then to LKB and XLE as the two most popular environments which rely on a graphical interface and high-level representations.

During unifications and applications of descriptions, TRALE’s SLD displays feature structures only on demand, and for one step at a time. This makes it hard to recognize these operations as processes and to understand them. Kahina’s GUI makes it easy to go back and forth between steps and to quickly compare different points in time. Neither does the old SLD provide explicit information on how and when constraints and procedural attachments are executed. Given the complexities of TRALE’s constraint formalism, this is a severe problem, since goals are often suspended until some preconditions are fulfilled, and are only then executed in a delayed fashion. In larger parses, this behavior makes it virtually impossible to infer the current state of execution from a linear trace. Kahina's two-dimensional step tree with specialized nodes for representing suspended goals makes these aspects much more transparent.

In a classical tracer, decisions are always made locally, without any possibility to correct errors. A single erroneous skip command or a small mistake in a breakpoint definition may force the user to abort and restart a long tracing process. This necessarily leads to defensive behavior to prevent the loss of relevant information. As a result, traces tend to take longer than they would if information on past steps remained accessible. Kahina improves the accessibility of non-local information by its support for post-mortem inspection, but also with the simultaneous graphical display of multiple types of information.

The other major debugging paradigm of grammar engineering is characterized by chart-based high-level debugging (LKB and XLE). The LKB is the most relevant point of comparison for a TRALE debugger, since both systems are primarily designed for HPSG implementations. The LKB excels at detailed error messages for violations of formal conditions, whereas for more complex debugging tasks, especially those involving rule interactions, its tools are a lot less developed. It is standard debugging procedure to find a short sentence that exhibits the relevant problem, and then to inspect the parse chart in a time-consuming process which may require substantial intuition about the grammar. Once the problem is isolated in a small set of phrases, LKB's mechanism for interactive unification checks comes into play. Any two structures in the feature structure visualization can be tested for unifiability. If unification fails, the user receives explicit feedback on the reasons for failure. To trace the interaction between multiple constraints, intermediate results of successful unifications are used to chain together unification checks.

While Kahina also supports high-level parse chart inspection in the spirit of the LKB, interactive unification is only supported experimentally. Kahina compensates for this by its much more direct support for locating sources of error. Since every single unification or retrieval step is fully exposed by the source-level debugger, the inefficient process of narrowing down a problem only by means of the chart and test parses can in most cases be avoided. This reduces the importance of interactive unification, since the relevant failure can already be observed in the full context of the original problematic parse.

The LFG parser of XLE consists of a c-structure parsing component, and a constraint system that subsequently enforces f-structure constraints. A display of legal c-structures for which no valid f-structures could be found provides more fine-grained feedback about the reasons for structure invalidity than in the LKB. The XLE chart shows edges for partial matches of c-structure rules. While this provides some of the desired information on failed edges, compared to Kahina it still lacks information on rules that fail to apply because already the first constituent cannot be established. For the failed edges that are shown, XLE provides advanced analysis tools, also for finding out why no valid f-structure for a given c-structure could be found. All views offer options for extending the displayed information by invalid or incomplete structures, and selecting such a structure will highlight the parts which were missing in a c-structure rule or which violated some f-structure constraint. The exact way in which f-structure constraints are enforced still remains intransparent. This means that XLE lacks Kahina's support for grammar optimization, because the order in which the individual constraints are enforced is not exposed and cannot be manipulated.

To sum up the discussion, Kahina combines desirable properties of both the chart-based and the tracer-based grammar debugging paradigms. The main advantage of its hybrid approach lies in providing support for high-level parse inspection via the chart interface, while still making it possible to find out and to visualize on demand how exactly a parse is computed, effectively giving direct and fine-grained access to sources of undesirable behavior. The procedural orientation also supports the application of profiling techniques for grammar optimization, which is not possible if access is limited to high-level abstractions.

A downside of Kahina's hybrid nature may be that it could take longer for the beginner to develop an efficient grammar debugging workflow than in other environments, mainly because heavy use of control agents is necessary to keep step data extraction efficient and manageable. Moreover, the high-level components do not provide very advanced functionality in the area of interactive error diagnostics by default. Users must dig a little deeper in search of error diagnosis, but the explanations they obtain along the way are very detailed and complete. Finally, Kahina's layered system architecture makes it straightforward to integrate further high-level error diagnosis modules tailored to the user requirements. The prototype of a grammar workbench presented in [5] gives a glimpse of related ongoing developments. Recent extensions are not yet completely integrated, and need to be evaluated by grammar developers before they can become part of a release.

8 Conclusion

Kahina is a framework for building debugging environments for different grammar implementation systems. Its debuggers can take advantage of a modular system architecture, predefined bridges for communication with Prolog systems, a variety of view components, and external software modules. An implementation time of less than 500 person hours for the most recent debugger, compared with an estimated 3,000 person hours for the first, demonstrates the effectiveness of the framework in facilitating the development of interactive graphical debuggers for additional grammar engineering systems. In the present paper, we emphasized the application to TRALE in our discussion of Kahina's strategies for breaking up traces into conceptually meaningful chunks of information. The amount of information presented to the user, and the strategies by which it is gathered (in small steps, forcing shortcuts, leaping with or without recording information) can be customized by means of control agents that offer a very powerful abstraction layer for modifying tracing behavior.

The design of Kahina is tailored to a hybrid approach to grammar debugging which attempts to combine the advantages of a high-level chart-based view of parsing processes with the usefulness of a Prolog tracer for understanding every aspect of computational behavior and system performance. Initial experiences with the new TRALE debugger indicate that its low-level components especially help novice users to gain better insight into controlling parsing behavior within the system. The needs of expert users are currently catered for by a flexible chart display with high interactivity. This aspect of the system will be strengthened in future work through the development of tools that go beyond the analysis of parses.

References

1. Byrd, L.: Understanding the control flow of Prolog programs. In: Tamlund, S.A. (ed.) Proceedings of the 1980 Logic Programming Workshop. pp. 127–138 (1980)
2. Carlsson, M., et al.: SICStus Prolog User's Manual, Release 4.1.1. Tech. rep., Swedish Institute of Computer Science (December 2009)
3. Carpenter, B., Penn, G.: ALE 3.2 User's Guide. Technical Memo CMU-LTI-99-MEMO, Carnegie Mellon Language Technologies Institute (1999)
4. Copestake, A.: Implementing Typed Feature Structure Grammars. CSLI Publications, Stanford, CA (2002)
5. Dellert, J.: Extending the Kahina Debugging Environment by a Feature Workbench for TRALE. Diplomarbeit, Universität Tübingen (February 2012)
6. Dellert, J.: Interactive Extraction of Minimal Unsatisfiable Cores Enhanced By Meta Learning. Diplomarbeit, Universität Tübingen (January 2013)
7. Dellert, J., Evang, K., Richter, F.: Kahina, a Debugging Framework for Logic Programs and TRALE. The 17th International Conference on Head-Driven Phrase Structure Grammar (2010)
8. Dewar, A.D., Cleary, J.G.: Graphical display of complex information within a Prolog debugger. International Journal of Man-Machine Studies 25, 503–521 (1986)
9. Eisenstadt, M., Brayshaw, M., Paine, J.: The Transparent Prolog Machine. Intellect Books (1991)
10. Evang, K., Dellert, J.: Kahina. Web site (2013), access date: 2013-03-04. http://www.kahina.org/
11. Kaplan, R., Maxwell, J.T.: LFG grammar writer's workbench. Technical report, Xerox PARC (1996), ftp://ftp.parc.xerox.com/pub/lfg/lfgmanual.ps
12. Lazarov, M.: Gralej. Web site (2012), access date: 2013-03-05. http://code.google.com/p/gralej/
13. Melnik, N.: From "hand-written" to computationally implemented HPSG theories. Research on Language and Computation 5(2), 199–236 (2007)
14. Pollard, C., Sag, I.A.: Head-Driven Phrase Structure Grammar. The University of Chicago Press, Chicago (1994)
15. Rumpf, C.: Offline-Compilierung relationaler Constraints in QType. Online slides (2002), access date: 2013-03-07. http://user.phil-fak.uni-duesseldorf.de/ rumpf/talks/Offline-Compilierung.pdf
16. Wielemaker, J.: An overview of the SWI-Prolog programming environment. In: Mesnard, F., Serebenik, A. (eds.) Proceedings of the 13th International Workshop on Logic Programming Environments. pp. 1–16. Katholieke Universiteit Leuven, Heverlee, Belgium (2003)