Kahina, a Debugging Framework for Logic Programs and TRALE

Johannes Dellert, Kilian Evang, Frank Richter
Seminar für Sprachwissenschaft
Eberhard Karls Universität Tübingen
{jdellert,kevang,fr}@sfs.uni-tuebingen.de

1 Introduction

We present a new graphical debugging framework, *Kahina*, for logic programming and demonstrate two of its key features by applying it to debugging HPSG grammars in TRALE (Penn et al., 2003). Debugging TRALE grammars, such as those presented in the textbook by Müller (2007), that stay true to their declarative specification in HPSG is a particularly challenging logic programming task, due to the size and complexity of these grammars. Kahina visualizes the structure of the parse process as a tree and relates each action to the grammar's source code, making it possible to maintain an overview over a large number of steps. The debugger lets users inspect each parse step with its complete context (variable bindings, executed constraints on feature structures, and effects of procedural attachments), thereby providing all information necessary to understand complex interactions of different grammar components that are otherwise very hard to discern. A novel and versatile breakpoint system based on tree automata permits precise searches for specific events during parsing.

2 Kahina as a Graphical Debugging Framework for Logic Programming

Graphical debugging in logic programming dates back to the 1980s and early 1990s (Dewar and Cleary, 1986; Rajan, 1986; Eisenstadt et al., 1991), with early systems focusing on teaching Prolog's execution model using small toy programs. One of the few freely available mature tools of today, SWI-Prolog's GUI tracer (Wielemaker, 2003), is restricted to displaying the current call stack at each point of execution without facilities for graphical post-mortem analysis. In recent years, interest has shifted to comparable tools for the more complex case of constraint logic programming, leading to the development of systems such as CLPGUI (Fages et al., 2004). The goal of the Kahina framework is to fill the gap of a technologically up-to-date graphical debugging framework for the logic programming paradigm, with large-scale symbolic grammar development as its initial target application.

In Kahina's modular architecture (Fig. 1) communication between a logic programming system and the core debugging system is implemented by a *bridge*. The bridge translates tracer output and other data provided by the logic program into Kahina's internal format, hands back control information to the logic program and defines application-specific data-types, such as TRALE's feature structures, as well as specialized view components to display them.

The view components currently available for TRALE are a source display, a feature structure display, a chart display and variable watching. While each of them only shows the respective details of *one* step at a time, any step can be selected at any time since Kahina stores the complete execution

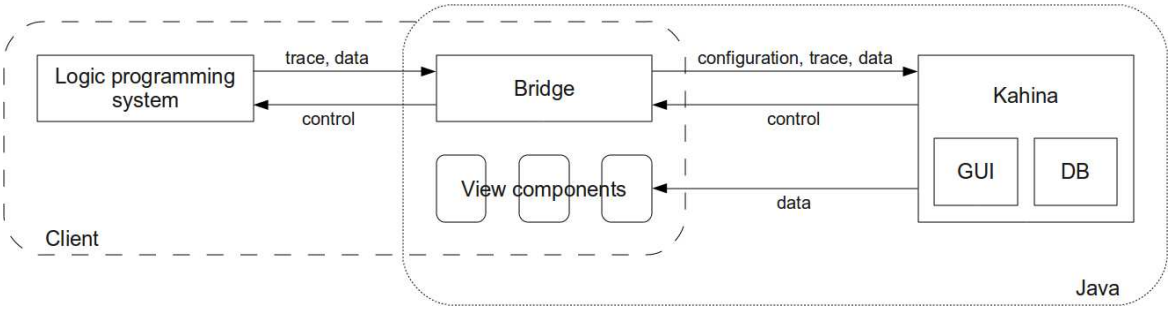

Figure 1: Architecture of a debugging system using the Kahina framework

history. This makes it possible to check for nonlocal effects arising from the interaction of complex constraints, edges on the parse chart, and logic programs attached to grammar rules or constraints.

Understanding and navigating complete execution histories is supported by two components with major innovations: a tree visualization of the execution space and a novel breakpoint system. In the remainder of this paper we focus on these two components and their application to TRALE.

3 Visualizing Parsing Processes Using Kahina

TRALE's parser follows the procedural box model (Byrd, 1980), breaking the parsing process down into meaningful units called *steps*. We conceptualize the execution space of a parsing process as a tree in which each node represents a step. Within the execution of procedural attachments and functional descriptions, steps closely correspond to invocations of ALE procedures and clauses. Elsewhere TRALE defines its own set of step types for actions such as (a) applying descriptions to feature structures (unification, adding types, feature selection) and (b) parsing (applying a rule, or closing an edge under rule application). Each tree node is labeled with the numeric ID and the type of the corresponding step. Each step can be selected by clicking on the corresponding node, thereby changing the contents of all relevant view components.

Parsing even a short sentence can easily comprise several thousand steps. Although conceptually all steps are part of a single tree, this huge tree would hardly be navigable. Our strategy to overcome this problem is to break down the whole tree into manageable pieces that represent meaningful units of execution. At the highest level, the graphical debugger splits it into two trees, an *overview tree* and a *detail tree*. The overview tree is a thinned-out version of the complete tree, eliminating all nodes except the "cornerstones" of a parsing process, *viz.* rule applications and closing edges under rule applications. By selecting individual cornerstone nodes, the user controls which part of the tree is shown as detail tree. The detail tree contains only the descendants of the currently selected cornerstone node, and only down to any lower cornerstone nodes, which become leaves.

Another challenge besides the size of execution spaces of logic programs is their complex structure. There are at least two tree structures in which the steps could be arranged, and which are both meaningful and important. The *search tree* shows the effects of backtracking. Without backtracking, a search tree is a long unary branch in which each step is a child of the step that was last invoked before it. On backtracking, the step that is the last active choicepoint is copied to represent the new invocation. The copy becomes a sibling of the original, creating a new branch. The *call tree* corresponds more closely to the logic program itself. The descendants of each step are those which are invoked in the course of its execution. In Prolog terminology, the children of a goal are its subgoals.

Kahina integrates the search tree and the call tree into one visualization by giving the tree diagrams in the graphical debugger a macrostructure and a microstructure (see Fig. 2). The macrostructure is the search tree. Here, the vertical visual axis shows the order in which steps are invoked within

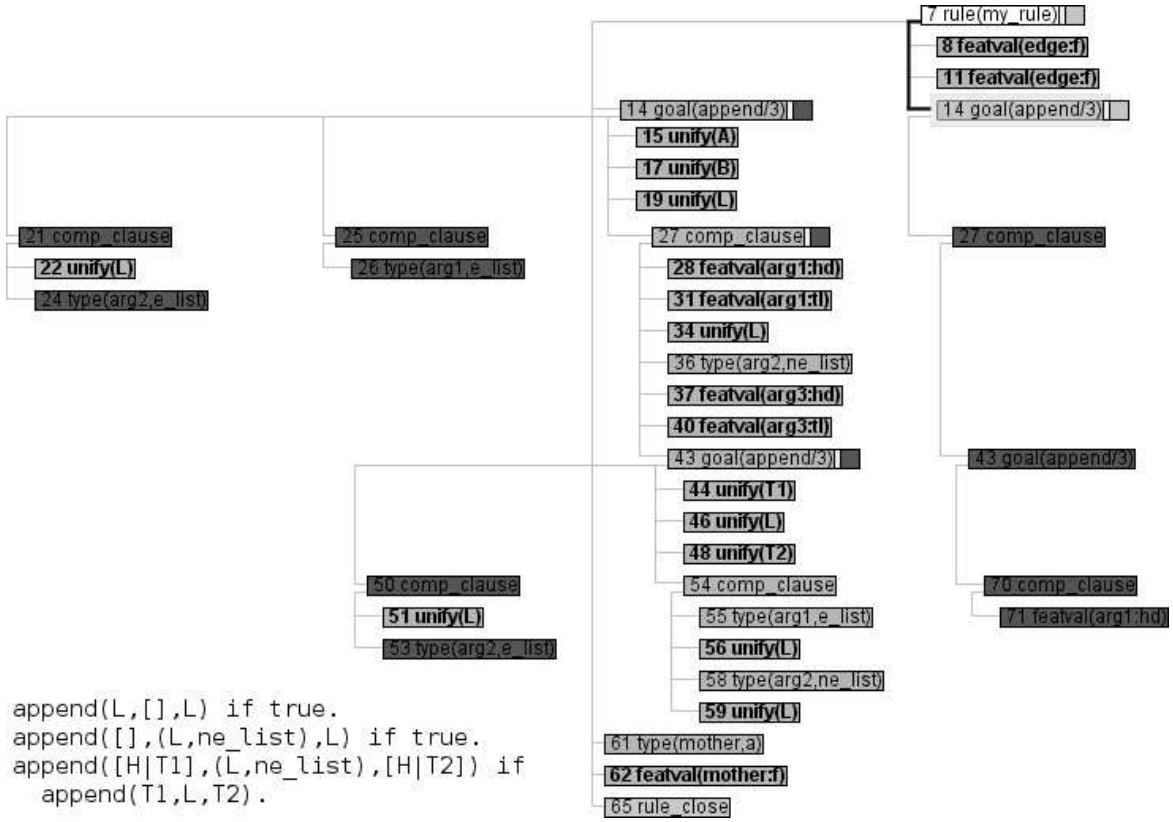

Figure 2: An ALE procedure for appending lists and a fragment of a detail tree showing its application to two lists with one element each. Steps in dark gray have failed, steps in light gray have successfully exited. The descendants of step 65 are not shown. The second copy of step 14 (on the right) is being redone and about to fail since a substep (27) has failed.

one branch. The horizontal axis places alternative branches next to each other, arranged from left to right in chronological order. Thus, the order of step invocations can be read off the diagram by doing an intuitive preorder traversal of the macrostructure. The microstructure of the detail tree uses the horizontal axis in another way: Node indentation and edgy lines connecting steps to their substeps represent the call tree. This layout results in a very transparent visualization as it corresponds closely to the way the head and body of clauses in Prolog and TRALE are usually formatted.

4 Breakpoints

Like a classical Prolog tracer, Kahina offers four basic commands for stepping through the execution of a logic program, or a parsing process: `creep` completes the current step or advances to the next step, `fail` allows to shortcut the execution of a step which will fail anyway or to explore otherwise inaccessible parts of the search space, `skip` automatically completes the current step without showing substeps (but stores them for later inspection), and `leap` advances execution to the next step where a certain condition, defined by the user in advance and called a *breakpoint*, holds.

In a classical Prolog tracer (e.g. Carlsson et al. (2009)), breakpoints are defined using various specialized types of breakpoint conditions such as unifiability of the current goal with a given term. Such tracers maximally have access to the backtrace, which does not include information from previously failed branches. This can be a severe limitation especially in environments such as TRALE's parser,

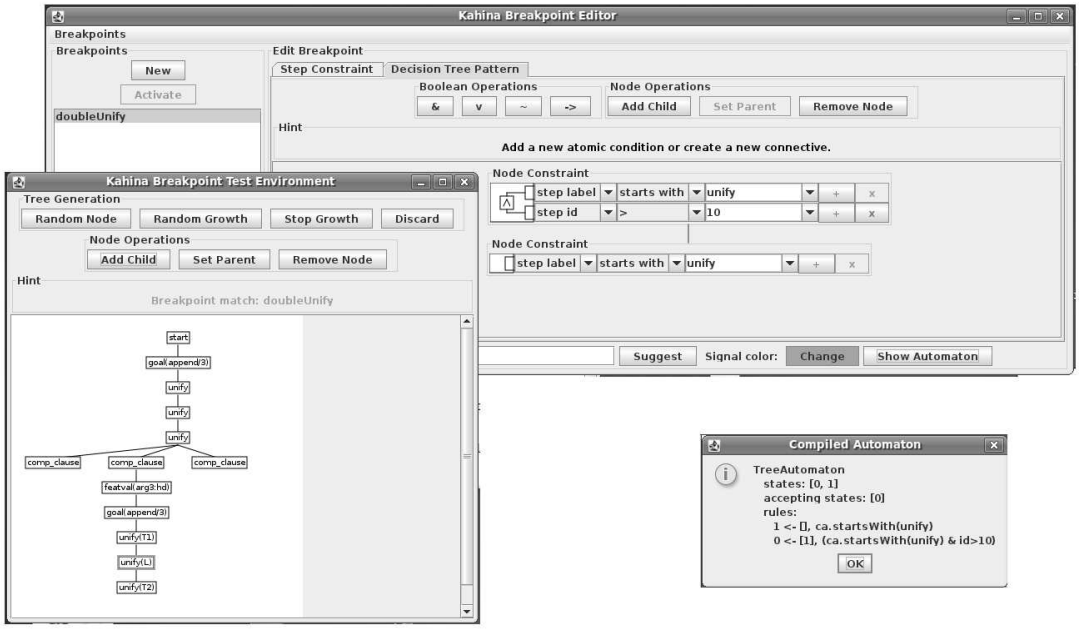

Figure 3: Prototype version of the breakpoint editor; the user is interested in a call of type `unify` which was caused by another call of type `unify` after step 10.

which makes heavy use of failure-driven loops and dynamic predicates.

Kahina's very general approach to defining breakpoints draws on the tree model of the execution space described in the previous section, which includes all information even about failed branches. In Kahina, breakpoints are arbitrary patterns on the two-dimensional step-tree structure. They are not restricted to being used with the `leap` command during execution, but also function as search patterns on completed trees after execution.

The added conciseness and expressive power of Kahina's extended breakpoint definitions has many advantages. Not every instance of arriving at a given source code location may be of interest, but only situations in which arriving there is the result of some particular predicate call which also caused a certain other goal to be evaluated. To capture such patterns, Kahina's breakpoint mechanism can define a tree fragment consisting of three source code matches, and the breakpoint will then only be lept to if arriving at the three locations was related in the way specified in the tree fragment.

Kahina comes with an intuitive graphical breakpoint editor for defining tree fragment patterns. The definition of simple patterns does not require a full understanding of the underlying formalism and can be achieved with a few mouse clicks and key strokes. Templates make defining certain very common types of patterns even easier. To help non-specialists, the breakpoint editor comprises a testing environment in which new users can check the effect of their breakpoints. In essence, the testing environment is a tree editor which mimics the construction of a step tree, enhanced by facilities for randomly generating large trees. Any number of breakpoints may be tested in parallel. Fig. 3 illustrates important functions of the breakpoint editor.

Internally, the tree fragment patterns are compiled into a variant of non-deterministic bottom-up tree automata that is specialized for efficient pattern matching in dynamically changing trees. We chose tree automata for this purpose because they constitute a well-understood and efficient formalism for pattern matching in trees, equivalent to regular expressions and finite automata in the domain of string pattern matching (see Comon et al. (2007) for a thorough introduction). A *breakpoint tree automaton* monitors a tree structure of Kahina steps by assigning *states* to the tree's leaves and then

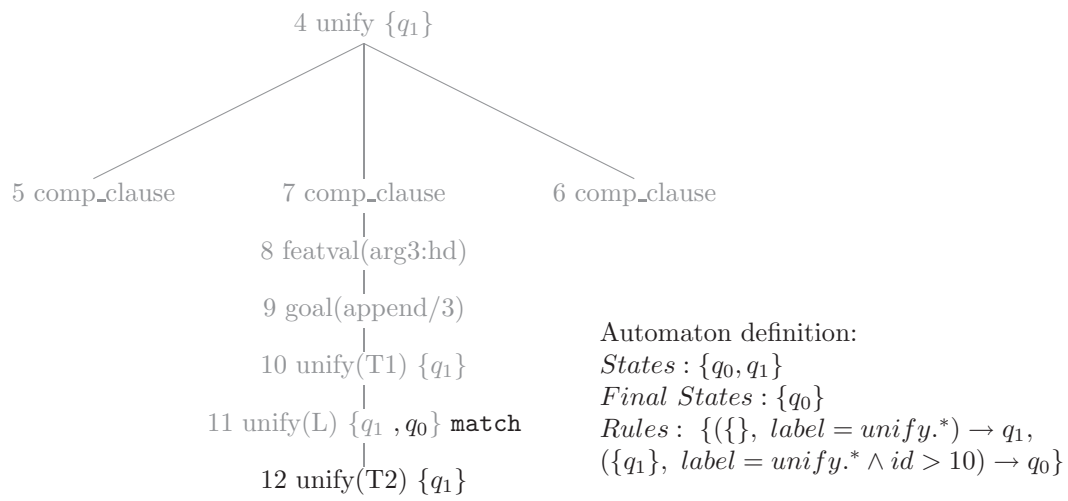

Figure 4: Adapting annotations after adding a new node

deducing, via a transition function, the set of states for each tree node from its label and the set of its child nodes' states. It is important to note that the transition rules have a subset semantics for child node states, i.e. a rule applies if the set of states given on the left-hand side of the rule is a *subset* of the union of the sets of states for its child nodes. Various customizable ways of defining whether a node symbol matches a rule are available, e.g. regular expressions on node labels and numeric constraints on node IDs.

A node *matches* the breakpoint defined by a breakpoint automaton as soon as recursive application of the transition function annotates it with one of the final states. This behavior differs from the usual definition of a tree automaton in that it does not only accept matches at the root node. Instead it detects the *occurrence* of a tree pattern in a tree, and a node recognized to match the breakpoint corresponds to the root of the tree pattern. Similar ideas were first proposed by Neumann and Seidl (1998) in the context of detecting matching subtrees in a single traversal of a static forest.

If we conceive of the states of the automaton at a given tree node as *annotations*, the updating process necessary after a change to the tree could be called a *reannotation*. Essentially, processing the addition of, or a change to, a tree node only requires the (re)annotation of the node itself, with subsequent recursive reannotation of its ancestor nodes. The process terminates once the root is reached or when reannotating a node does not change the set of its annotations. Fig. 4 shows what happens internally after the last update process that leads to the match during the breakpoint test in Fig. 3. The gray part of the tree represents the situation before the new node with label unify(T2) was added, the result of the reannotation process is highlighted in black.

5 Conclusion

Kahina is a graphical debugging framework that can be of great help in handling complex logic programming tasks like the implementation of HPSG grammars in TRALE. Its graphical viewers support grammar developers in one of their hardest tasks, systematically tracking bugs and inefficiencies in grammars. The intuitive and detailed visualization considerably eases the memory burden involved in keeping track of huge parsing processes, including intricate interactions between rules, constraint applications, and suspended goals in co-routining.

Tree automata as a formalism for the breakpoint system add a powerful pattern matching mechanism on call structures to the grammar developer's toolbox for quick and efficient exploration of large parsing processes. The analysis can even be conducted post-mortem, since all potentially relevant

details of the parsing process are stored in a data base for later retrieval and inspection.

The innovations introduced by Kahina thus offer new solutions to challenges that can make debugging grammars very time-consuming, and, in certain situations and without appropriate high-level tools, practically impossible. Future enhancements of the system will include profiling facilities and gradual evolution towards an integrated development environment for debugging TRALE and other constraint-based parsing systems.

References

Lawrence Byrd. Understanding the control flow of Prolog programs. In S-A Tamlund, editor, *Proceedings of the 1980 Logic Programming Workshop*, pages 127–138, 1980.

Mats Carlsson et al. SICStus Prolog User's Manual, Release 4.1.1. Technical report, Swedish Institute of Computer Science, December 2009.

H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

Alan D. Dewar and John G. Cleary. Graphical display of complex information within a Prolog debugger. *International Journal of Man-Machine Studies*, 25:503–521, 1986.

Marc Eisenstadt, Mike Brayshaw, and Jocelyn Paine. *The Transparent Prolog Machine*. Intellect Books, 1991.

François Fages, Sylvain Soliman, and Rémi Coolen. CLPGUI: a generic graphical user interface for constraint logic programming. *Journal of Constraints*, 9(4):241–262, 2004.

Stefan Müller. *Head-Driven Phrase Structure Grammar. Eine Einführung*. Stauffenburg, Tübingen, 2007.

Andreas Neumann and Helmut Seidl. Locating matches of tree patterns in forests. In *Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science*, pages 134–145. Springer, 1998.

Gerald Penn, Bob Carpenter, and Mohammad Haji-Abdolhosseini. *The Attribute Logic Engine User's Guide with TRALE Extensions*, December 2003. Version 4.0 Beta.

Tim Rajan. APT: A principled design for an animated view of program execution for novice programmers. Technical Report 19, Human Cognition Research Laboratory, The Open University, Milton Keynes, 1986.

Jan Wielemaker. An overview of the SWI-Prolog programming environment. In Fred Mesnard and Alexander Serebenik, editors, *Proceedings of the 13th International Workshop on Logic Programming Environments*, pages 1–16, Heverlee, Belgium, 2003. Katholieke Universiteit Leuven.