Multimedia Tools and Applications (2022) 81:299–317
https://doi.org/10.1007/s11042-021-10624-6

1191: SENTIENT MULTIMEDIA SYSTEMS

Visual exploration of visual parser execution

Gennaro Costagliola[1] · Mattia De Rosa[1] · Vittorio Fuccella[1] · Mark Minas[2]

Received: 22 June 2020 / Revised: 3 November 2020 / Accepted: 29 January 2021 /
Published online: 18 March 2021

Abstract

In this paper, we present ParVis, an interactive visual system for the animated visualization of logged parser trace executions. The system allows a parser implementer to create a visualizer for generated parsers by simply defining a JavaScript module that maps each logged parser instruction into a set of events driving the visual system interface. The result is a set of interacting graphical/text windows that allows users to explore logged parser executions and helps them to have a complete understanding of how the parser behaves during its execution on a given input. We used our system to visualize the behavior of textual as well as visual parsers and describe here two of these uses. Moreover, in order to validate the efficacy of our system, we ran a user experiment where students analyzed a CUP-generated parser both with ParVis, instantiated to LALR parsers, and the standard CUP debug facilities. The results show that students can indeed analyze parser behavior and find mistakes in parser specifications more easily and quickly using ParVis. In particular, in some parser design tasks, using ParVis participants achieved a higher success rate of 50% in 42% less time with respect to the baseline system.

Keywords Parser visualization · Visual parsing · Graph parsing · Visualization · Program visualization · Human-computer interaction

✉ Mattia De Rosa
matderosa@unisa.it

Gennaro Costagliola
gencos@unisa.it

Vittorio Fuccella
vfuccella@unisa.it

Mark Minas
mark.minas@unibw.de

[1] Department of Informatics, University of Salerno, Via Giovanni Paolo II, Fisciano, SA, Italy

[2] Computer Science Department, Universität der Bundeswehr München, Neubiberg, Germany

1 Introduction

Parsing textual or visual inputs has always been considered a not easy task both for researchers working on the definition and/or automatic generation of new parsers, and students facing their compiler classes. In fact, since the 80s, in order to ease their work, researchers and teachers have produced and used many tools able to visualize the data structures and behavior for a multitude of parsing and parser generation algorithms. These tools are mostly ad-hoc and very specific to the chosen algorithm to visualize so that they cannot be used or extended to visualize other parsers.

In order to overcome this, we built ParVis, a visual system for the animated visualization of logged parser trace executions. The system is relatively easy to extend for the creation of new parser visualizers. This is done by simply defining a JavaScript module that maps each logged parser instruction into a set of events driving the visual system interface.

We have used ParVis to visualize the behavior of many parsers, in particular, LR, GLR-based and top-down parsers for visual languages (VL parsers, in short), derived by both hypergraph [10] and positional grammars [9], and LALR(1) parsers as generated by the well-known parser generator CUP [36]. In the case of VL parsers (a significant part of VL research in the last 25 years [6]), the system has been used to help researchers in having useful feedbacks while building the corresponding parser generators. In the LALR case, the system has been used to help students to better understand how LALR parsing works and to have insights on the correctness of both the grammar they are writing and its input.

The system adds to the many approaches that have been studied and implemented in the last years. In our case, ParVis has been built with the aim to create a system providing a simple interface to parser implementers with application to any type of parsers including VL parsers.

A prototype for visualizing the execution trace of one hypergraph based parser has already been presented in [7, 8]. However, as opposed to ParVis, it does not support adaptation to other parsers.

In this paper, we describe two uses of ParVis: one for CUP-generated LALR parsers and the other one for GPSR-generated parsers, which are Visual GLR-based parsers based on hyperedge replacement grammars. For the first instance, we give the description and the results of a complete usability experiment drawn on 14 computer science students who took the *Compilers* class. The efficacy of the use of ParVis in building visual environments to trace parsers' execution turned out to be very high.

This work is an extension of [5]. The new contribution consists of the GPSR ParVis instance described in Section 3.2 and the user experiment described in Sections 4 and 5.

The paper is organized as follows: Sections 2 and 3 describe the architecture of ParVis and its instances, respectively, while Sections 4 and 5 report the empirical evaluation of the system on CUP-generated parsers and its results. The related work is described in Section 6 and the final conclusions are given in Section 7.

2 ParVis architecture

ParVis has been built to create an interactive tool to visualize and explore the behavior of a parser when executed for a given input. We considered a simple animated replay was not sufficient, because parsers are generally quite complicated and several data structures correspond to each other during execution (in an LALR parser, e.g., the parse stack, the partial derivation trees and the input text processed so far). Instead, the tool should allow the user to playback the parser behavior in an animated way, to stop the animation, to execute it step by step, to rewind or to jump to any point of the parser execution. The different data structures should be able to be displayed as graphs (e.g. derivation trees) and texts (e.g. the input to be processed) in different views. The cross-references between the different views should be visualized interactively, for example, by clicking in one view and highlighting it accordingly in another view.

On the one hand, the tool should be general enough to be used with as many parsers as possible and to support as many different data structures as possible. On the other hand, it should be easy to use in order to help to better understand the sometimes complicated behavior of parsers by supporting domain-specific representations and showing their changes during parser execution in an appealing animated way.

Another requirement results from the fact that most parsers are generated by parser generators. Manually customizing such parsers so that their behavior can be visualized with PARVIS would be tedious. Instead, one would prefer to extend the parser generator so that it automatically generates such parsers. A further requirement, therefore, is that existing parser generators should be easy to extend in this way.

PARVIS implements these goals with the architecture shown in Fig. 1. The visualization of the parser behavior is done offline after the termination of the parser. The parser must be able to create a log file about its execution, which is the basis for the later visualization. Therefore the log file must inform about the sequence of the executed analysis steps, but also about the processed input file.

PARVIS represents the parser execution as a sequence of so-called events for which a Java API is provided. Instead of an unstructured sequence of events, they can be structured as an event tree, which is described in more detail below.

A log file must be translated into an event tree to visualize them. This is done by the reader script, which is programmed in JavaScript and executed by the script interpreter. It reads the log file and creates the event tree in memory using the Java API.

After completion of building the event tree, the PARVIS runtime visualizes and animates the parser execution in the PARVIS user interface. The UI provides the user with several controls for this purpose.

Based on this architecture, two steps are necessary to adapt PARVIS to a parser generator or the parsers generated by it and to visualize the behavior of these parsers with PARVIS:

1. One has to extend the parser runtime in a way that a log file is generated during each parser run.
2. One has to write the reader script that translates each log file into an event tree.

These steps are only necessary once for a parser runtime. PARVIS can then visualize the behavior of all parsers that use this runtime (e.g. all parsers generated with the parser generator CUP).

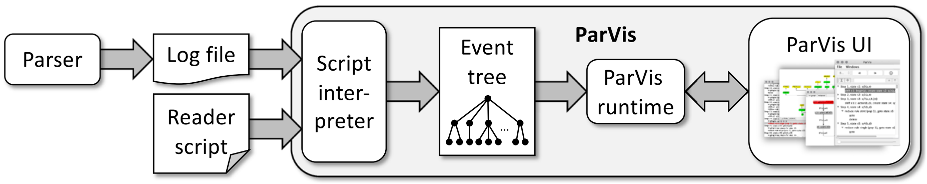


Fig. 1 PARVIS architecture

Basically, there is no restriction for the format used in log files. Since the reader script is programmed in JavaScript, JSON is easy to process. However, XML or any other text format is also possible.

2.1 Events and event tree

Each animated visualization consists of a sequence of events. For illustration, let us consider the following example, which is a sequence of three events: The first event (e_1) creates a node in the view that represents the parse stack, the second event (e_2) creates an edge from this node to the previously added node, and the third event (e_3) deletes the last created node with its outgoing edge.

If such event sequences are automatically processed one after the other, an animation is created. However, it is also possible to execute these events step by step or to go back step by step to return to earlier visualization states. Furthermore, one can jump to any event or fast forward or rewind the visualization accordingly.

ParVis provides a Java API for the creation of a large number of such events. Each event is represented by an object in memory, which can be processed forward or backward. To visualize the event sequence e_1, e_2, e_3 in the above example, the associated event objects must be processed forward one after the other. If one stops processing and then goes back one step, the event object for event e_3 must be processed backwards, that is, the node that was deleted previously and its outgoing edge must be recreated and displayed in the view.

Parsers perform their analysis as a sequence of analysis steps, for example, in an LALR parser, as a sequence of shift and reduce steps, which in turn can be divided into a sequence of actions. For example, a reduce step first removes a certain number of states from the parse stack and then pushes a new state on the stack. Each of these actions, in turn, consists of a sequence of events that correspond to a visualization. Besides primitive events (e.g. the creation of new nodes or edges), ParVis also provides composite events. They each combine a sequence of events into a higher-level one. For example, to process a composite event, the events grouped together below it must be processed one after the other. The sequence of all events of a parser execution can thus be structured hierarchically in the form of the so-called *event tree*. This allows one to represent the individual analysis steps and the actions "contained" in them as composite events in a LALR parser. Actions can then consist of sequences of primitive events.

Figure 2 shows an example to illustrate event trees and their generation from the log file using the reader script. Figure 2a shows a log entry of a log file in JSON format. It describes that a GPSR parser (see Section 3.2) has created a new state $q_1(a, n)$. This entry is processed in the reader script by the `state` function (Fig. 2b). The parameter `obj` then contains the object after the JSON object has been read. The function creates a sequence of primitive events by instantiating the corresponding event classes of the Java API. First, an event is created to create a new node. The parameters provided indicate that the node is to be created in the graph `gss` and should belong to the sequence of events under the composite event `curAction`. Please note that by executing `new Node` the reader script actually does not create a node in `gss`, but an event. The node is only created when this event is executed during the processing of the event tree. And if ParVis has to process this event backwards later, this node is deleted again.

The remaining lines show that further events are created, including Tooltip and Click-Decoration events, which provide the node with a tooltip when executed or register activities that ParVis performs when the user clicks on the node in the view.

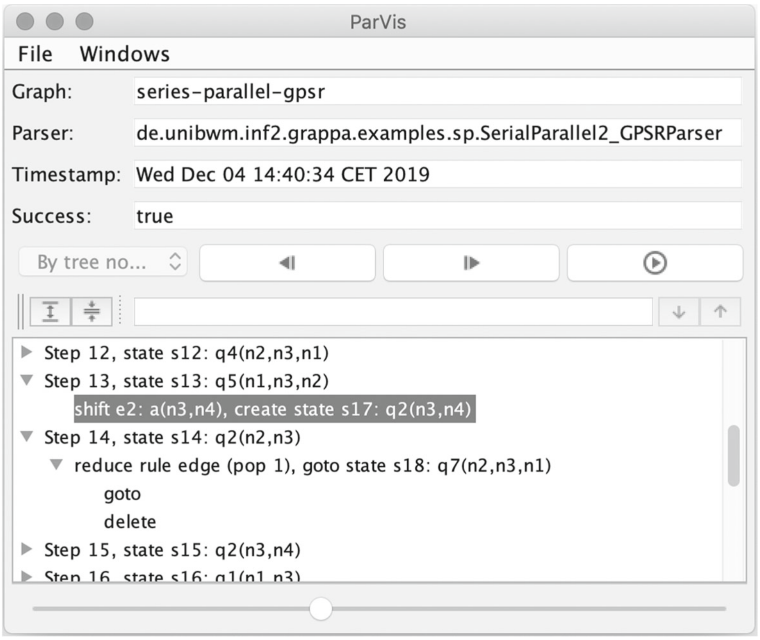
a)
```
{
   "op": "state",
   "id": "s2",
   "number": 1,
   "shortName": "q1",
   "longName": "q1(a,n)",
   "consumedNodes": [ "a", "n" ],
   "consumedEdges": ["e1" ],
   "children": [ "s1" ]
}
```

b)
```
function state(obj) {
   var s = new Node(obj.longName, gss, initialPos(gss, obj.children), curAction);
   event[obj.id] = s;
   new NodeStyle(s, defaultStateColor, curAction);
   new NodeLabel(s, obj.id + ": " + obj.longName, NodeLabelType.CENTER, curAction);
   var tooltip = "<html>State: " + obj.id + " " + obj.longName +
                 "<br>Consumed nodes: " + obj.consumedNodes +
                 "<br>Consumed edges: " + obj.consumedEdges + "</html>";
   new TooltipDecoration(s, tooltip, curAction);
   new ClickDecoration(s, unhighlightAll, curAction);
   new ClickDecoration(s, highlightGSSObject(s), curAction);
}
```

Fig. 2 Fragments of a log file in JSON format (**a**) and of a reader script (**b**)

2.2 ParVis user interface

The ParVis UI only has the ParVis main window as a fixed component. It shows information about the log file and the event tree in which one can collapse composite events (see Fig. 3). Clicking on one of these events rewinds or moves forward the visualization to this



Graph: series-parallel-gpsr
Parser: de.unibwm.inf2.grappa.examples.sp.SerialParallel2_GPSRParser
Timestamp: Wed Dec 04 14:40:34 CET 2019
Success: true

▶ Step 12, state s12: q4(n2,n3,n1)
▼ Step 13, state s13: q5(n1,n3,n2)
  shift e2: a(n3,n4), create state s17: q2(n3,n4)
▼ Step 14, state s14: q2(n2,n3)
  ▼ reduce rule edge (pop 1), goto state s18: q7(n2,n3,n1)
    goto
    delete
▶ Step 15, state s15: q2(n3,n4)
▶ Step 16, state s16: q1(n1,n3)

Fig. 3 ParVis main window

event. There are also VCR-like controls for executing the next or previous event as shown in the event tree. In addition, one can use the play button to automatically process the sequence of events displayed in the tree as an animated visualization one after the other.

Further windows can be set up by the reader script. These include windows for displaying graphs. ParVis uses the graph drawing library yFiles[1] for the automatic layout of graphs and smooth animations. yFiles provides a variety of automatic layout methods, e.g. hierarchical layout (layered graph drawing), tree layout, spring embedder. The layout method can also be specified via the reader script. Text windows can also be created. Controlled by events, text parts can be highlighted (see the following section).

In this way, ParVis can be easily configured for different parsers. In the following, we demonstrate ParVis instances for two parsing approaches.

3 ParVis instances

ParVis has already been used by the authors to visualize the execution of LALR parsers generated with CUP[2] and visual language parsers. Examples of the latter are *Predictive Top-Down* (PTD) [10], *Predictive Shift-Reduce* (PSR) [11, 12], and *Generalized PSR* (GPSR) parsers [15, 25] generated with Grappa[3] as well as pLR parsers [9]. As described in the previous section, we had to extend the parser runtime and write a reader script, so that the parser generates a log file during its execution, which translates the script into an event tree. The log file format used is JSON apart from the pLR case where XML was used instead.

In the following, we refer to ParVis, which can display the log files of a family of parsers by means of a single reader script, as a (customized) ParVis *instance*.

All these ParVis instances have in common that, for reasons of clarity, the parser data structures are visualized in several windows. However, the representation in the different windows is never independent of each other. Instead, the user can interactively explore the relationships between the separately displayed data structures. This is made possible by the fact that selecting components in one window leads to highlighting the associated components in another window. This functionality must be implemented by the respective reader script. ParVis provides an API for this.

In the following, we describe two ParVis instances, one for LALR parsers generated by the parser generator CUP, and one for hypergraph parsers generated by the parser generator Grappa. In particular, we discuss how they use interaction and highlighting to visualize the relationship between the information distributed to different windows.

3.1 CUP

CUP (Construction of Useful Parsers) is an LALR parser generator written in Java that implements standard LALR(1) parser generation [36]. In order to be able to use ParVis, we have modified the CUP runtime so that the parsers generated by it are able to also write a log.json file during the parsing of an input file. This job has been simplified by the fact that CUP already produces log messages that could be easily used to produce the JSON log.

[1] https://www.yworks.com/products/yfiles

[2] http://www2.cs.tum.edu/projects/cup/

[3] http://www.unibw.de/inf2/grappa

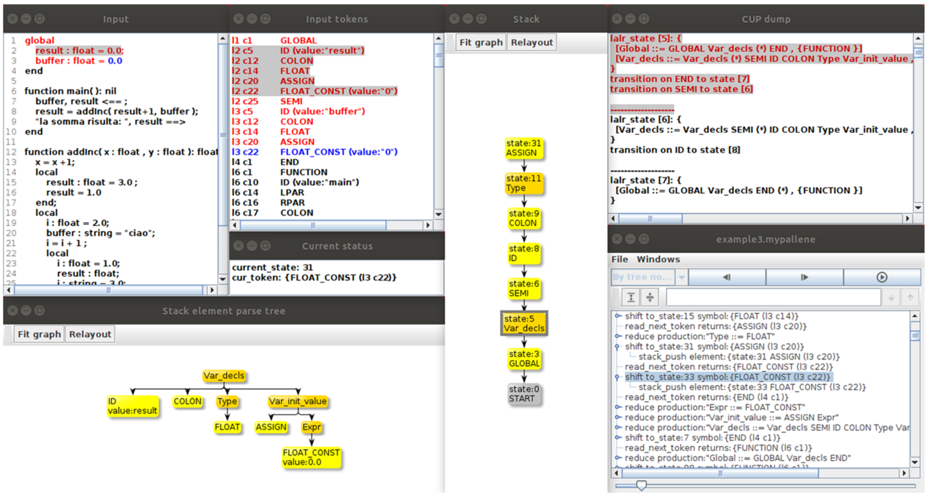
The log file lists the classic steps performed by an LALR parser: read the next token from the input, reduce with a grammar production, shift to a state, syntax error, accept input. Each step "contains" its component operations: push/pop on the stack, goto another state of the automaton, etc.

In addition to the main ParVis window, the ParVis instance for CUP consists of 6 windows, as shown in Fig. 4, that are mainly used to show the current status of the parser (i.e. the status before the execution of the highlighted event on the main window). They are listed in the following, from left to right and from top to bottom.

– *Input*: shows the parsing input file. The already processed input is shown in red, the last read (but still unprocessed) token is in blue, and the rest is in back.
– *Input tokens*: shows the tokenization of the input file. For each token, line and column in the input file are shown, and the same colors of the *Input* window are used.
– *Stack*: shows the current contents of the stack as a sequence of graphical nodes. Yellow nodes correspond to states that have just processed a terminal, while orange nodes to states that have just processed a non-terminal. In addition, a tooltip shows the kernel of the sets of items of the corresponding state for each node.
– *CUP dump*: shows the information produced by CUP during the generation of the parser, which includes the lists of: the states of the automaton, the grammar terminals, the grammar non-terminals, the grammar productions.
– *Current status*: shows the current state number and the last token read from the input.
– *Stack element parse tree*: shows the parsing tree underlying a stack node in the *Stack* window, when the node is left-clicked. The same colors are used for the nodes in the *Stack* window. In addition, a tooltip on non-terminals shows the number of the grammar production used to generate it.

Numerous user interactions are also possible, and in particular for each window:

- *Input tokens*: a left-click on a token updates all the windows (except the *Stack element parse tree* window) to the state immediately after the execution of the *read_next_token*



```
global
   result : float = 0.0;
   buffer : float = 0.0
end

function main( ): nil
   buffer, result <== ;
   result = addInc( result+1, buffer );
   "la somma risulta: ", result ==>
end

function addInc( x : float , y : float ): float
   x = x +1;
   local
      result : float = 3.0 ;
      result = 1.0
   end;
   local
      i : float = 2.0;
      buffer : string = "ciao";
      i = i + 1 ;
      local
         i : float = 1.0;
         result : float;
```

```
l1 c1     GLOBAL
l2 c5     ID (value:"result")
l2 c12    COLON
l2 c14    FLOAT
l2 c20    ASSIGN
l2 c22    FLOAT_CONST (value:"0")
l2 c25    SEMI
l3 c5     ID (value:"buffer")
l3 c12    COLON
l3 c14    FLOAT
l3 c20    ASSIGN
l3 c22    FLOAT_CONST (value:"0")
l4 c1     END
l6 c1     FUNCTION
l6 c10    ID (value:"main")
l6 c14    LPAR
l6 c16    RPAR
l6 c17    COLON
```

```
current_state: 31
cur_token: {FLOAT_CONST (l3 c22)}
```

```
lalr_state [5]: {
  [Global ::= GLOBAL Var_decls (*) END , {FUNCTION }]
  [Var_decls ::= Var_decls (*) SEMI ID COLON Type Var_init_value ,
}
transition on END to state [7]
transition on SEMI to state [6]

------------------
lalr_state [6]: {
  [Var_decls ::= Var_decls SEMI (*) ID COLON Type Var_init_value ,
}
transition on ID to state [8]

------------------
lalr_state [7]: {
  [Global ::= GLOBAL Var_decls END (*) , {FUNCTION }]
}
```

```
shift to_state:15 symbol:{FLOAT (l3 c14)}
read_next_token returns:{ASSIGN (l3 c20)}
reduce production:"Type ::= FLOAT"
shift to_state:31 symbol:{ASSIGN (l3 c20)}
   stack_push element:{state:31 ASSIGN (l3 c20)}
read_next_token returns:{FLOAT_CONST (l3 c22)}
shift to_state:33 symbol:{FLOAT_CONST (l3 c22)}
   stack_push element:{state:33 FLOAT_CONST (l3 c22)}
read_next_token returns:{END (l4 c1)}
reduce production:"Expr ::= FLOAT_CONST"
reduce production:"Var_init_value ::= ASSIGN Expr"
reduce production:"Var_decls ::= Var_decls SEMI ID COLON Type Var
shift to_state:7 symbol:{END (l4 c1)}
read_next_token returns:{FUNCTION (l6 c1)}
reduce production:"Global ::= GLOBAL Var_decls END"
```

Fig. 4 ParVis instance for a CUP generated parser (after clicking the *state:5* node in the *Stack window*)

action that has returned the clicked token; a right-click on a token highlights the corresponding element in the *Input* window.

- *Input*: it has a similar behavior as the *Input tokens* window. A right-click on the text highlights the corresponding token in the *Input* window.
- *Stack*: a click on an element causes the following changes to the other windows:
  - *Input*: the input text processed up to the element state is highlighted.
  - *Input tokens*: the input tokens processed up to the element state are highlighted.
  - *CUP dump*: the set of items and their goto transitions of the corresponding element state are highlighted.
  - *Stack element parse tree*: the parse tree corresponding to the element symbol is shown.

  The *Fit graph* button makes the stack occupy the whole window (without exceeding it), while the *Relayout* button causes a new application of the layout algorithm.
- *Current status*: a click on the current token causes its highlighting in the *Input* and *Input tokens* windows.
- *Stack element parse tree*: a click on a tree node causes the corresponding consumed text and tokens to be highlighted, respectively, in the *Input* and *Input tokens* windows.

In order to validate this ParVis instance, we report an empirical evaluation and its results in Sections 4 and 5.

3.2 GPSR

This example demonstrates that ParVis can not only be used for conventional parsers that analyze text. Rather, one can also create ParVis instances for parsers based on hypergraph grammars that analyze hypergraphs and can thus be used for visual languages. Here the concepts of hypergraphs, hypergraph grammars, and their parsers will be described only so far as to be able to understand their ParVis instances. Details can be found, e.g., in [12, 14, 15, 25].

Hyperedge replacement grammars (HRGs) are special hypergraph grammars and a direct extension of context-free grammars, such as those underlying LALR parsers. They operate on hypergraphs. These are generalized graphs whose edges are called hyperedges and can connect not only two but, depending on the type of the hyperedge, any number of nodes. The type of a hyperedge is expressed by its label.

Figure 5 shows a graphical representation of a hypergraph: Nodes are represented by ellipses, hyperedges by rectangles. Each ellipse contains the name of the respective node, while each rectangle contains the label and thus the type of the respective hyperedge. Arrows between rectangles and ellipses indicate which nodes are visited by the hyperedges, numbers indicate the order of the visited nodes.

Note, however, that hypergraphs are mathematical structures without a fixed representation rule. Hypergraphs can serve as a mathematical model of visual languages whose syntax and structure are described by HRGs. In this example we consider flowcharts (see the top-left window in Fig. 6). Nodes are small black circles, hyperedges are yellow rectangles, green diamonds and orange rounded rectangles that represent actions, decisions or start and stop. Arrows indicate which nodes are connected by a hyperedge. The direction of arrows indicates control flow.

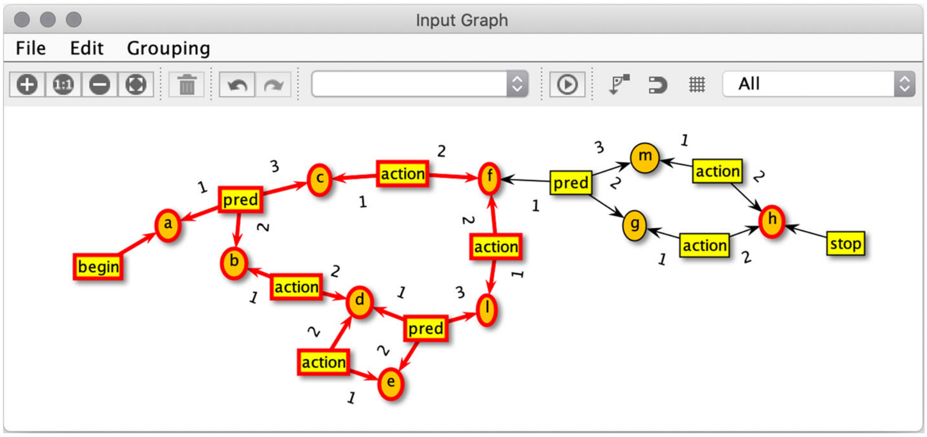


Fig. 5 Generic hypergraph representation

In fact, this flowchart and the hypergraph in Fig. 5 have the same hypergraph as a model. The label of each hyperedge in the hypergraph determines its shape in the flowchart, e.g. *pred*-hyperedges are represented as diamonds, which indicate decisions. That way, the flowchart and the hypergraph differ just in the way how hyperedges are depicted and in the overall layout: whereas the flowchart is drawn from top to bottom as usual, nodes and hyperedges in Fig. 5 are arranged almost arbitrarily (in fact, a simple spring embedder layout has been employed).

ParVis must be able to graphically represent hypergraphs. Compared to the visualization of input texts for LALR parsers, an additional difficulty is that hypergraphs can be displayed in a variety of ways. Of course, a generic representation as shown in Fig. 5 is basically sufficient, but it does not represent the diagram modeled with the hypergraph sufficiently. Figure 6 illustrates that the domain-specific representation in the form of a flowchart is more suitable. The reader script creates this representation as a graph from the log information about the input hypergraph and then displays it in a ParVis window. The ParVis API provides various node and edge shapes and layout algorithms for this purpose. For instance, it also allows adding tooltips with detailed information on the represented hyperedge. Figures 5 and 6 show the results of this transformation (and a tooltip example). The generic representation (Fig. 5) is always available; the specific flowchart representation in Fig. 6 naturally assumes that the input graph models a flowchart. Different types of diagrams then require different reader scripts.

Like context-free grammars, HRGs use replacement rules, which replace hyperedges carrying non-terminal labels. The derivation process and derivation trees are defined similarly as for context-free grammars. We consider *Generalized Predictive Shift-Reduce* (GPSR) parsers [15, 25] here, which "lift" the idea of textual *Generalized LR* (GLR) parsers[4] to HRGs and which can be used for any HRG. GPSR parsers manage not just one, but many stacks simultaneously, which are stored and processed in a so-called *graph-structured stack* (GSS) compressed in the form of a DAG. The top-right window in Fig. 6 shows such a GSS:

[4]GLR parsers are in fact an extension of LR parsers [34].

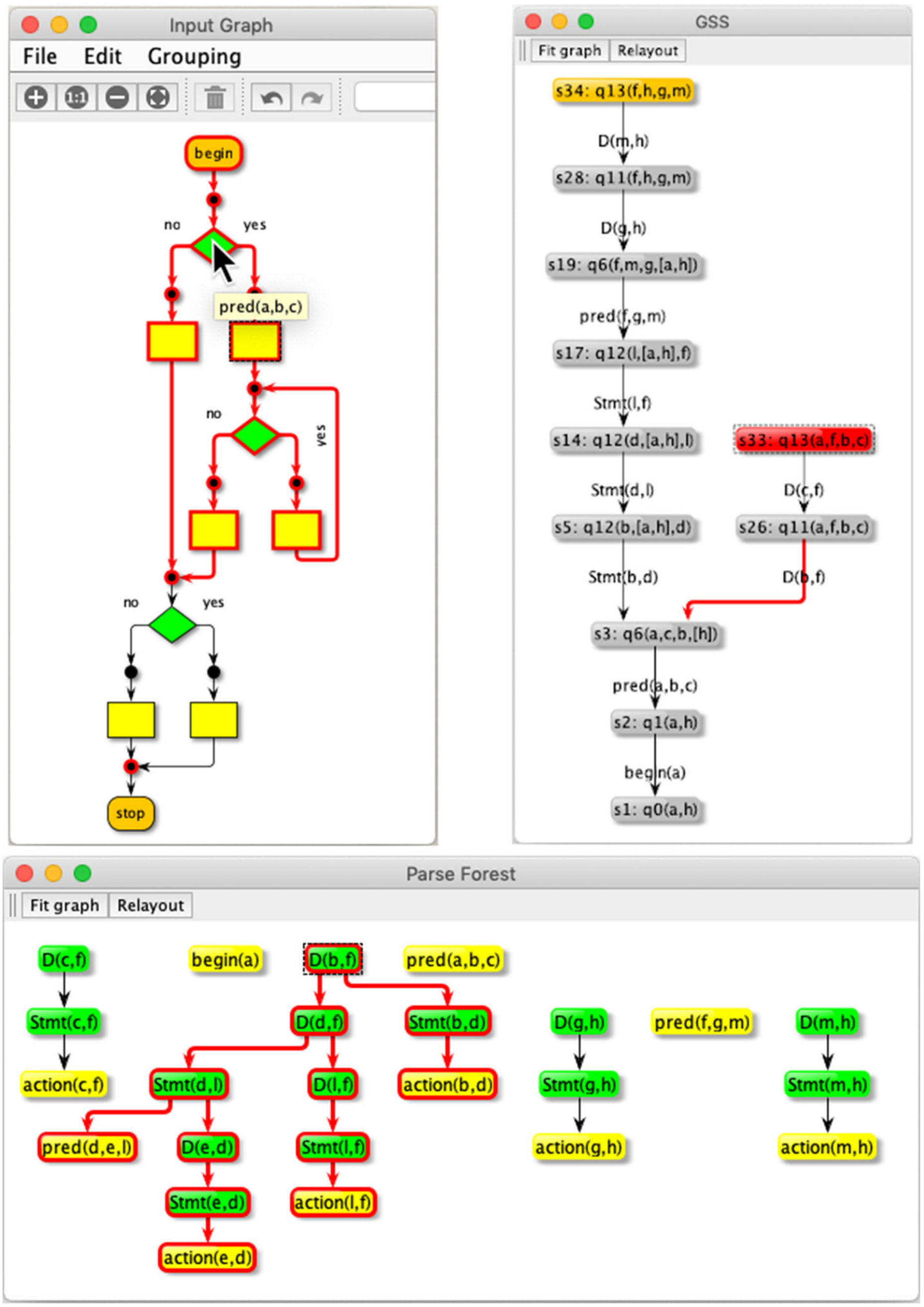


Fig. 6 ParVis instance of a GPSR parser for flowcharts

The nodes of this GSS are states of the parser, edges refer to predecessor states. We will not go into the exact meaning of these states of their labels here. Details can be found in [12, 15].

In fact, each path from a top state to the unique bottom state corresponds to one of the stacks processed simultaneously. The GSS in Fig. 6, therefore, represents two stacks simultaneously. Compare the stack representation in Fig. 4, which always consists of exactly one path. The edges of the GSS are labeled with hyperedges, which have been pushed onto the stack together with the subsequent state.

The topmost states of a stack are highlighted in orange or red. The current state processed by the parser is shown in red. If the user clicks on a state or if the parser has selected a state for current processing during execution and it is therefore displayed in red, all nodes and hyperedges of the input hypergraph already processed in this state are highlighted (see the red highlights in the flowchart representation in Fig. 6 or in the generic hypergraph representation in Fig. 5). In this way, one can see the order in which nodes and edges of the hypergraph are processed during the parser execution. Note that, in contrast to texts, hypergraphs do not have an inherently predefined processing sequence.

In a separate window (on the bottom of Fig. 6) the partial derivation trees identified by the parser so far are displayed as a *parse forest*.[5] The roots of these trees correspond to the edge labels in the GSS. However, this correspondence is not permanently visualized in order not to overload the display. Instead, the corresponding derivation tree is highlighted when the user clicks on an edge of the GSS, or conversely, the edge of the GSS is highlighted when the user clicks on the root of a derivation tree.

The ParVis instance with the generic representation of hypergraphs, which can be used for all GPSR parsers, contains 388 lines of JavaScript code. The domain-specific flowchart representation requires an additional 64 lines. The scripts read log files in JSON format, which are generated by the GPSR parser runtime.

Without a corresponding visualization of the GPSR parsers, the development of the GPSR parser generator as part of Grappa would hardly have been possible. One of the concretely solved problems was the deletion of states from the GSS. In contrast to LALR parsers, in which each reduce step performs a fixed number of pop operations and thus automatically removes states from the stack, the situation in the GSS is more complicated. Pop operations on a stack represented in the GSS may only delete states if they are not simultaneously a member of other stacks. Without suitable visualization of the graphical structure of the GSS, the realization of this functionality would have been very difficult, if at all.

In fact, parser visualization had a major impact on the research on GPSR parsers: Original GPSR parsers [15] turned out to be too inefficient without manual tuning. This is so because GPSR parsers identify parses of an input graph in a search process that may run into dead ends. They are inefficient because they waste time in this process and because they discard all information collected in these dead ends. They rather recreate the same information over and over again. The details of these problems remained hidden in the complexity of the parsers' data structures, but was easy to see after using an earlier prototype of ParVis and visualizing the progress of GPSR parsers. Memoization techniques [25] then helped to overcome these problems, which would not have been identified without parser visualization.

[5]GPSR parsers make in fact use of so-called compressed parse forests similar to GLR parsers.

4 Evaluation

We carried out a user-study aimed at evaluating the performance of the PARVIS instance for CUP-generated parsers from both the points of view of effectiveness and efficiency. To this aim, we compared it to a baseline development environment (see Section 4.2). In the experiment, we asked participants to use both systems to complete some simple and common tasks in parser design.

4.1 Participants

For the experiment, we recruited 14 (all male) participants. They were all computer science master students between 22 and 27 years old ($M = 24.2$, $SD = 1.8$). All of them declared medium or high knowledge of parsers (they all had attended a course of compilers for their master's degree), all except one declared a medium or high knowledge of IntelliJ IDEA.[6] Eleven of them also had a medium or high knowledge of its debugging environment. Lastly, eleven of them had a medium or high ability to read and interpret the log file of CUP parser generator.

4.2 Apparatus

The experiment was conducted in a well-lit laboratory, with all participants doing the same task at the same time. To put the participants at ease, we allowed them to run the experiment directly on their laptops, on which many of them already had some software installed for the exam of compilers. Although the systems were heterogeneous (different operating systems and computational power), they all had the minimum requirements for running the experimental software.

The experimental software for the baseline condition was the IntelliJ IDEA IDE with its integrated debugging environment and CUP parser generator (our modified version already mentioned in Section 3.1). The screenshot in Fig. 7 shows the standard support provided to developers using CUP, that is the CUP log and debug messages (bottom) and the content of the XML file (resultTree.xml, top) produced when running the program.

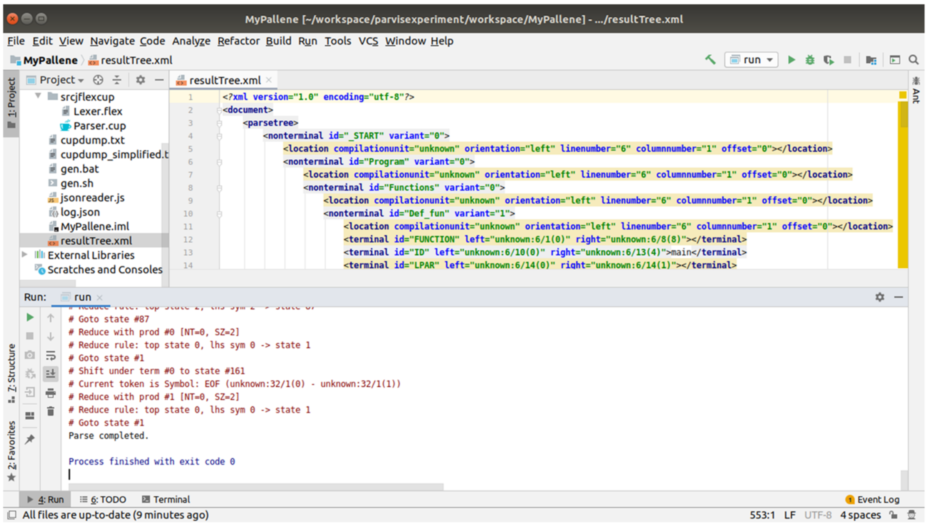
Additionally, in only one experimental condition, the participants were allowed to use PARVIS (see Section 3.1 for a description of the interface). Before starting the experiment, we made sure that all laptops were aligned with the same software bundle (except for the operating system).

4.3 Procedure

Before starting the experiment, participants were instructed on the aims and procedures of the experiment; they were then asked to complete a brief survey asking age, gender, level of knowledge of parsers and parser generators, programming IDEs and debugging environments.

Before starting the experiment, participants had a short (about 30 minutes) explanation on the use of PARVIS and a 30 minutes practice session to get familiar with its use to get information and solve simple parsing problems.

[6]https://www.jetbrains.com/idea/



```
<?xml version="1.0" encoding="utf-8"?>
<document>
    <parsetree>
        <nonterminal id="_START" variant="0">
            <location compilationunit="unknown" orientation="left" linenumber="6" columnnumber="1" offset="0"></location>
            <nonterminal id="Program" variant="0">
                <location compilationunit="unknown" orientation="left" linenumber="6" columnnumber="1" offset="0"></location>
                <nonterminal id="Functions" variant="0">
                    <location compilationunit="unknown" orientation="left" linenumber="6" columnnumber="1" offset="0"></location>
                    <nonterminal id="Def_fun" variant="1">
                        <location compilationunit="unknown" orientation="left" linenumber="6" columnnumber="1" offset="0"></location>
                        <terminal id="FUNCTION" left="unknown:6/1(0)" right="unknown:6/8(8)"></terminal>
                        <terminal id="ID" left="unknown:6/10(0)" right="unknown:6/13(4)">main</terminal>
                        <terminal id="LPAR" left="unknown:6/14(0)" right="unknown:6/14(1)"></terminal>
```

```
# Goto state #87
# Reduce with prod #0 [NT=0, SZ=2]
# Reduce rule: top state 0, lhs sym 0 -> state 1
# Goto state #1
# Shift under term #0 to state #161
# Current token is Symbol: EOF (unknown:32/1(0) - unknown:32/1(1))
# Reduce with prod #1 [NT=0, SZ=2]
# Reduce rule: top state 0, lhs sym 0 -> state 1
# Goto state #1
Parse completed.

Process finished with exit code 0
```

Fig. 7 IntelliJ IDEA IDE provided to developers using CUP

Each participant had to perform a total of two sessions (one for each experimental condition), each including four different tasks. As the tasks were knowledge-based (participants had to acquire knowledge and report a response) we designed tasks with two variants, a and b, in order to avoid knowledge transfer from the first experimental condition to the other. The tasks were the following:

1. discover the error in the grammar and report it (one wrong terminal was added in a production);
2. discover the error in the grammar and report it (one wrong non-terminal was added to a production);
3. discover the number of nodes in a parsing tree for a given input block and report it;
4. discover the states in the stack after an input token was processed and report them.

We set a time limit of 10 minutes to complete a task. All participants started the same task at the same time. At the end of a task, the participant had to submit the response through a simple web-form prepared on the e-learning platform of the department (at the end of the 10 minutes the form was automatically submitted containing what had been written by the participant up to that moment). The submission time was chosen as a timestamp for task completion. Participants who sent the response before the time limit could rest up to the start of the next task. At the end of each session, participants were allowed to rest for a few minutes. In tasks where text commentary was required for the response (not numeric response), its correctness was evaluated by the researchers.

After completing the two sessions, participants were asked to fill a System Usability Scale (SUS) [3] questionnaire for each of the two systems. SUS includes ten statements, that alternate between positive and negative, to which respondents have to specify their level of agreement using a five-point Likert scale. Each SUS questionnaire has a score between 0 and 100, which was averaged on all participants. Finally, they were asked to fill a questionnaire, in which they where asked their preferred system (PARVIS or *baseline*) and their feedback for the two systems in open form.

4.4 Design

The experiment was a single-factor within-subjects design. The factor was the *system*, which included two levels: ParVis or *baseline*. Our dependent variables were the *task success rate* and the *total task time*.

In the following, we refer to a *trial* as a single task execution by a single participant, and so the total number of trials was 14 participants $\times$ 4 tasks $\times$ 2 variants $=$ 112 trials. The task success rate was calculated as the fraction of the count of trials with a correct response over the total number of trials. The *total task time* was calculated as the total time (not including rests and breaks) spent on the four tasks.

We counterbalanced the two systems by randomly assigning a number (between 1 and 14) to each participant and arranging the sessions according to the order shown in Table 1.

Besides counterbalancing our single factor, the scheme reported in the table had the following advantages:

– it allowed our participants to switch just once between the two experimental conditions, thus changing their computer settings just once;
– it allowed us to have all participants doing the same task at the same time, thus simplifying the execution and control of the experiment.

5 Results and discussion

All participants completed the experiment. For each participant, the experiment lasted about one hour and a half, including rests and excluded training. As our dependent variables are ratio scale measurements, we tested significance on both of them using a paired-samples t-test.

5.1 Task success rate

Regarding the *task success rate*, the grand mean was 53.6%. ParVis had a higher success rate of 64.3%, while baseline had 42.9%, with an advantage for ParVis of 50.0%. Figure 8 reports the task success rates in detail for all tasks and systems.

The effect of *system* on *task success rate* was statistically significant ($t(13) = -2.747$, $p = .0166$).

5.2 Total task time

Regarding the *total task time* (reported here in minutes and seconds), the grand mean was $16'53''$. Participants were faster with ParVis, for which we recorded an average time spent on the four task of $12'23''$, while for *baseline* we had $21'23''$. This gives ParVis a time

Table 1 Counterbalancing scheme used in the experiment

Participants	Session 1 Tasks T1(a) T2(a) T3(a) T4(a)	Session 2 Tasks T1(b) T2(b) T3(b) T4(b)
1-7	ParVis	*baseline*
8-14	*baseline*	ParVis

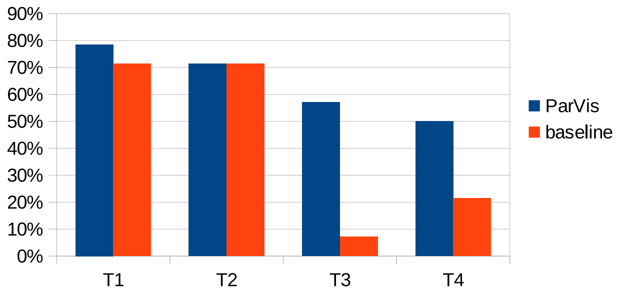


Fig. 8 Task success rate

saved of 42.1%. For more details, Fig. 9 reports the average time spent on each task for each system.

The effect of *system* on *session time* was statistically significant ($t(13) = 4.716$, $p < .001$).

5.3 User satisfaction and free-form comments

As regards user satisfaction, the mean SUS score was 81.8 ($SD = 12.8$) for ParVis and 35.0 ($SD = 15.0$) for *baseline*. As SUS scores are supposed to be an interval scale measurement [31], we performed a paired-samples t-test on them. The test revealed a statistical significance between the two techniques ($t(13) = -8.910$, $p < .001$). This result was confirmed by the final questionnaire, where all participants preferred ParVis.

From the open-feedback questionnaire, we recorded appreciation for ParVis from many participants. In particular, some comments were enthusiastic: "wished I had this before"; "very fast and intuitive"; "the software is simple to use despite the complexity of the parsing operations. The system offers an overview of the parsing output, making the final situation clear and simplifying the bug check and fix operations". From comments on the baseline system, we understood that ParVis allows overcoming limitations of the baseline system, e.g. the difficulty in understanding the debug messages given by CUP during parsing and in realizing the content of the stack and the execution flow.

Most complaints about ParVis were due to the high number of windows opened at the same time which were difficult to manage for some participants. They would have preferred a single-window interface.

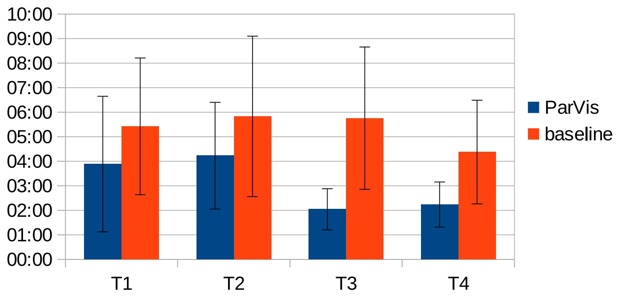


Fig. 9 Average time spent on each task. Error bars show the standard deviation

5.4 Discussion

The first two tasks were defined in such a way that students could easily find the solution in a few minutes with either of the two systems. Whereas the task success rates were very similar in both cases, task completion time was considerably less with PARVIS. The remaining two tasks were made intentionally more challenging in order to stress the use of the two systems. The third task required to discover a parse tree after a certain amount of the input had been processed. This was especially more difficult for students using *baseline*; they had to interpret the CUP dump information, which contains parse trees as condensed XML structures. Apparently, this is more difficult and time consuming than using parse tree visualizations provided by PARVIS. This has been confirmed by the fact that only a single participant was able to complete this task successfully using the baseline system. The last task assessed the capability of tracing the states on the stack while parsing the input. In this case, 50% of the students using PARVIS, but only 21% of the students using *baseline* were able to present a solution in time.

It should be mentioned that most of the students were in fact familiar with the grammar used in the experiment; they had been working on this grammar during the previous month when they developed their compiler project. Moreover, they were already familiar with the programming environment consisting of CUP and IntelliJ IDEA, which they used during their Compiler course. Nevertheless, participants performed better using PARVIS.

6 Related work

Data structure and algorithm visualizations have been studied for more than 35 years [4, 20, 33], and now many web resources exist implementing visualizations and animations of almost all the most common data structures and algorithms based on them, e.g. Algomation[7] and VisulAlgo.[8] However, lately, the research in this field has shrunk considerably. Most of the important papers are in the range from 1980 to 2000 and the applications have been basically two: visual debugging [26] and teaching and learning [32]. Among the still currently developed tools is JSAV [20], a JavaScript algorithm visualization library that is meant to support the development of general algorithm visualizations for online learning material.

Visualization tools more specific to the case of parsers can be categorized in *parsing algorithm visualizers* and *visual compilers*.

Parsing algorithm visualizers are generally able to illustrate the construction of a parser from a grammar, and then to show the execution of the created parser on a provided input string. These tools often do not allow to export the generated parser since they have been built mostly for teaching purposes. Some examples are: LLparse and LRparse [2] and [21] that visualize predictive parsing and simple LR parsing; JFLAP [29], still in development, that is capable of displaying recursive-descent parsing, predictive parsing, simple LR parsing, and CYK parsing; Predictive Parsing Visualization Tool (PPVT) [16], a recent tool, that is capable of showing the functioning of predictive parsers; and Parsing Algorithms Visualization Tool (PAVT) [30].

[7] http://www.algomation.com

[8] https://visualgo.net

A *visual compiler* visually shows its execution to the user during the compilation of a program (usually with speed control left to the user), showing structures such as symbol table, parsing stack and parse tree. The first tools on PL/0 [1] and CLANG [27] were hand-coded and date back to the 1980s, whereas, in the following decade, the focus was on modified compiler-compilers to generate them. These include the tools Visual YACC [35], GYACC [23], CUPV [19], VOCOCO [28], LISA [24]. In particular, CUPV is also based on CUP but discontinued. It used to provide a GUI to the generated parser showing the parser data structures (e.g. stack, etc.). It could also optionally visualize reductions, individual item sets, lookaheads and semantic values (for which the user could implement dedicated views, for example, for an Abstract Syntax Tree).

Among other tools we can cite PAT [13, 17], now discontinued, which has been used for the visualization and statistical comparison of various GLR parsers, [18] for visualizing lexical generation processes and [22] that is an educational tool for visualizing compiling techniques based on deterministic parsers.

In all the tools above, the visualization code is embedded either in the parser or in its parser generator, requiring a big development effort and making them very specific to the implementation.

In this categorization, ParVis can be considered as a visual compiler with the exception that the visualization is based on a logged list of parser execution steps. This allows a developer to easily create a new parser visualization tool for a given parser or parser generator, with the cost of adding code for the generation of the log (besides its *reader script*). As shown in Section 3.1, this has been fairly easy with CUP.

7 Conclusions and future work

In this paper, we presented a new architecture for the easy development of parser visualization tools. The architecture can be applied to any type of parser and requires as input a log file containing a trace of execution of the considered parser and a JavaScript file for its interpretation. We introduced two instances of ParVis: one to visualize the execution of CUP generated parsers, for educational purposes, and one to visualize the behavior of a generalized hypergraph-based parser to support the researchers during its development. In order to validate the efficacy of the use of ParVis we also ran a usability study by involving 14 compiler students and presented the results that were very encouraging.

As future work, we plan to add statistical comparison features to ParVis in order to support comparisons among the parser instances created. Furthermore, we plan to add the possibility to run a parser and its ParVis instance concurrently in order to have online visualization of parser execution.

Funding Open access funding provided by Università degli Studi di Salerno within the CRUI-CARE Agreement.

References

1. Andrews K, Henry RR, Yamamoto WK (1988) Design and implementation of the UW illustrated compiler. In: Proceedings of the ACM SIGPLAN 1988 conference on programming language design and implementation, PLDI '88. Association for Computing Machinery, New York, pp 105–114. https://doi.org/10.1145/53990.54001
2. Blythe SA, James MC, Rodger SH (1994) LLParse and LRparse: visual and interactive tools for parsing. SIGCSE Bull 26(1):208–212. https://doi.org/10.1145/191033.191121
3. Brooke J (1996) SUS: a 'quick and dirty' usability scale. In: Jordan PW, Thomas B, Weerdmeester BW, McClelland IL (eds) Usability evaluation in industry, chap. 21. Taylor and Francis, London, pp 189–194
4. Brown MH, Sedgewick R (1985) Techniques for algorithm animation. IEEE Softw 2(01):28–39. https://doi.org/10.1109/MS.1985.229778
5. Costagliola G, De Rosa M, Fuccella V, Minas M (2020) ParVis: a visual tool for exploring parser execution traces. In: Proceedings of the 2020 international conference on advanced visual interfaces, AVI '20. Association for Computing Machinery, New York. https://doi.org/10.1145/3399715.3399853
6. Costagliola G, De Rosa M, Fuccella V, Perna S (2018) Visual languages: a graphical review. Inf Vis 17(4):335–350. https://doi.org/10.1177/1473871617714520
7. Costagliola G, De Rosa M, Minas M (2019) Visual parsing and parser visualization. In: 2019 IEEE symposium on visual languages and human-centric computing (VL/HCC). IEEE Computer Society, Los Alamitos, pp 243–247
8. Costagliola G, De Rosa M, Minas M (2020) Visualizing visual parser execution. In: Proc. of the 26th int. DMS conf. on visualization and visual languages, pp 57–66. https://doi.org/10.18293/DMSVIVA20-013
9. Costagliola G, Deufemia V, Polese G (2004) A framework for modeling and implementing visual notations with applications to software engineering. ACM Trans, Softw Eng Methodol 13(4):431–487. https://doi.org/10.1145/1040291.1040293
10. Drewes F, Hoffmann B, Minas M (2015) Predictive top-down parsing for hyperedge replacement grammars. In: Parisi-Presicce F, Westfechtel B (eds) Graph transformation, 8th international conference, ICGT 2015, L'Aquila, Italy, Lecture Notes in Computer Science, vol 9151, pp 19–34. Springer International Publishing. https://doi.org/10.1007/978-3-319-21145-9_2
11. Drewes F, Hoffmann B, Minas M (2017) Predictive shift-reduce parsing for hyperedge replacement grammars. In: de Lara J, Plump D (eds) Graph transformation: 10th international conference, ICGT 2017, held as part of STAF 2017, Marburg, Germany, July 18-19, 2017, Proceedings, Lecture Notes in Computer Science, vol 10373, pp 106–122. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-61470-0_7
12. Drewes F, Hoffmann B, Minas M (2019) Formalization and correctness of predictive shift-reduce parsers for graph grammars based on hyperedge replacement. Journal of Logical and Algebraic Methods in Programming 104:303–341. https://doi.org/10.1016/j.jlamp.2018.12.006. Preprint available also at arXiv:1812.11927[cs.FL]
13. Economopoulos GR (2006) Generalized LR parsing algorithms. Ph.D. thesis, Royal Holloway, University of London, UK. http://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.435358
14. Habel A (1992) Hyperedge replacement: grammars and languages, Lecture Notes in Computer Science, vol 643. Springer, Berlin. https://doi.org/10.1007/BFb0013875
15. Hoffmann B, Minas M (2019) Generalized predictive shift-reduce parsing for hyperedge replacement graph grammars. In: Martín-Vide C, Okhotin A, Shapira D (eds) Language and automata theory and applications, 13th international conference, LATA 2019, St. Petersburg, Russia, March 26-29, 2019, Proceedings, Lecture Notes in Computer Science, vol 11417, pp 233–245. Springer International Publishing, Cham. https://doi.org/10.1007/978-3-030-13435-8_17
16. Jain A, Goyal A, Chakraborty P (2017) PPVT: a tool to visualize predictive parsing. ACM Inroads 8(1):43–47. https://doi.org/10.1145/3002136
17. Johnstone A, Scott E, Economopoulos G (2004) The grammar tool box: a case study comparing GLR parsing algorithms. Electronic Notes in Theoretical Computer Science 110:97–113. https://doi.org/10.1016/j.entcs.2004.06.008
18. Jorgensen A, Economopoulos GR, Fischer B (2011) VLEx: visualizing a lexical analyzer generator - tool demonstration. In: Language descriptions, tools and applications, LDTA 2011, Saarbrücken, Germany, March 26-27, 2011. Proceeding, pp 12. https://doi.org/10.1145/1988783.1988795
19. Kaplan A, Shoup D (2000) CUPV–a visualization tool for generated parsers. In: Proceedings of the Thirty-first SIGCSE technical symposium on computer science education, SIGCSE '00. Association for Computing Machinery, New York, pp 11–15. https://doi.org/10.1145/330908.331801

20. Karavirta V, Shaffer CA (2013) JSAV: the JavaScript Algorithm visualization library. In: Proceedings of the 18th ACM conference on innovation and technology in computer science education, ITiCSE '13. Association for Computing Machinery, New York, pp 159–164. https://doi.org/10.1145/2462476.2462487
21. Khuri S, Sugono Y (1998) Animating parsing algorithms. In: Proceedings of the twenty-ninth SIGCSE technical symposium on computer science education, SIGCSE '98. Association for Computing Machinery, New York, pp 232–236. https://doi.org/10.1145/273133.274303
22. Krebs N, Schmitz L (2014) Jaccie: a Java-based compiler-compiler for generating, visualizing and debugging compiler components. Sci Comput Program 79:101–115. https://doi.org/10.1016/j.scico.2012.03.001
23. Lovato ME, Kleyn MF (1995) Parser visualizations for developing grammars with Yacc. In: Proceedings of the twenty-sixth SIGCSE technical symposium on computer science education, SIGCSE '95. Association for Computing Machinery, New York, pp 345–349. https://doi.org/10.1145/199688.199855
24. Mernik M, Zumer V (2003) An educational tool for teaching compiler construction. IEEE Trans Educ 46(1):61–68. https://doi.org/10.1109/TE.2002.808277
25. Minas M (2019) Speeding up generalized PSR parsers by memoization techniques. In: Echahed R, Plump D (eds) Proceedings tenth international workshop on graph computation models, Eindhoven, The Netherlands, 17th July 2019, *Electronic Proceedings in Theoretical Computer Science*, vol 309, pp 71–86. Open Publishing Association. https://doi.org/10.4204/EPTCS.309.4
26. Mukherjea S, Stasko JT (1994) Toward visual debugging: integrating algorithm animation capabilities within a source level debugger. ACM Trans Comput-Hum Interact 1:215–244
27. Resler D (1990) Visiclang–a visible compiler for clang. SIGPLAN Not 25(8):120–123. https://doi.org/10.1145/87416.87483
28. Resler RD, Deaver DM (1998) VCOCO: a visualisation tool for teaching compilers. In: Proceedings of the 6th annual conference on the teaching of computing and the 3rd annual conference on integrating technology into computer science education: changing the delivery of computer science education, ITiCSE '98. Association for Computing Machinery, New York, pp 199–202. https://doi.org/10.1145/282991.283123
29. Rodger SH, Finley TW (2006) JFLAP: an interactive formal languages and automata package. Jones & Bartlett Learning
30. Sangal S, Kataria S, Tyagi T, Gupta N, Kirtani Y, Agrawal S, Chakraborty P (2018) PAVT: a tool to visualize and teach parsing algorithms. Educ Inf Technol 23(6):2737–2764. https://doi.org/10.1007/s10639-018-9739-x
31. Sauro J, Lewis JR (2012) Standardized Usability Questionnaires. In: M. Kaufmann (ed) Quantifying the user experience, pp 185–240. https://doi.org/10.1016/B978-0-12-384968-7.00008-4
32. Shaffer CA, Cooper ML, Alon AJD, Akbar M, Stewart M, Ponce S, Edwards SH (2010) Algorithm visualization: the state of the field. Trans Comput Educ 10(3):9:1–9:22. https://doi.org/10.1145/1821996.1821997
33. Stasko JT (1990) Simplifying algorithm animation with Tango. In: Proceedings of the 1990 IEEE workshop on visual languages, pp 1–6. https://doi.org/10.1109/WVL.1990.128374
34. Tomita M (1985) An efficient context-free parsing algorithm for natural languages. In: Proceedings of the 9th international joint conference on artificial intelligence - vol 2, IJCAI'85, pp 756–764. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA. http://dl.acm.org/citation.cfm?id=1623611.1623625
35. White EL, Ruby J, Deddens LD (1999) Software visualization of LR parsing and synthesized attribute evaluation. Software: Practice and Experience 29(1):1–16. https://doi.org/10.1002/(SICI)1097-024X(199901)29:1⟨1::AID-SPE216⟩3.0.CO;2-N. https://onlinelibrary.wiley.com/doi/abs/10.1002/7-024X
36. Wilhelm R, Seidl H, Hack S (2013) Compiler design: syntactic and semantic analysis. Springer Publishing Company, Incorporated