Automatic and Dynamic Visualization of Process-Based Concurrent Programs

Daniel Augusto Rodrigues Farina ✉
Polytechnic Institute of Bragança, Portugal
Federal University of Technology - Paraná, Brazil

Rodrigo Campiolo ✉
Federal University of Technology - Paraná, Brazil

José Rufino ✉
Polytechnic Institute of Bragança, Portugal

Maria João Varanda Pereira ✉
Polytechnic Institute of Bragança, Portugal

Abstract

This article discusses the development of *forkSim*, a tool that can be used to support the teaching of system-level programming within the context of Operating Systems classes, by facilitating the comprehension and analysis of the behavior of C codes representing process-based concurrent programs involving *fork* system calls. The tool builds on two main components. The first is a C preprocessor, created using language processing techniques. This preprocessor embeds inspectors into the C code before its execution. In runtime, the inspectors extract relevant data from the actions performed and generate a JSON file. The second component is a web application that generates a visual representation of the program flow based on the JSON file. This visualization incorporates elements from BPMN diagrams and draws inspiration from representations used for many years in OS classes. The development of *forkSim* faced several technical challenges and involved some design decisions, both documented in this paper, along with a discussion of the results achieved.

2012 ACM Subject Classification Software and its engineering → Parsers; Computing methodologies → Concurrent computing methodologies; Human-centered computing → Visualization systems and tools

Keywords and phrases E-Learning Tool, System Programming, Concurrent Programming, Code Instrumentation, Inspector Functions, C, JSON, Konva JS, Canvas, Python, React JS, BPMN

Digital Object Identifier 10.4230/OASIcs.SLATE.2024.6

Funding This work was supported by national funds through FCT/MCTES (PIDDAC): CeDRI, UIDB/05757/2020 (DOI:10.54499/UIDB/05757/2020) and UIDP/05757/2020 (DOI: 10.54499/UIDP/05757/2020); and SusTEC, LA/P/0007/2020 (DOI: 10.54499/LA/P/0007/2020).
Daniel Augusto Rodrigues Farina: Research Centre in Digitalization and Intelligent Robotics (CeDRI), Laboratório para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC), Instituto Politécnico de Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal.
Rodrigo Campiolo: Federal University of Technology - Paraná (UTFPR), Department of Computer Science, Campo Mourão, Paraná, Brazil.
José Rufino: Research Centre in Digitalization and Intelligent Robotics (CeDRI), Laboratório para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC), Instituto Politécnico de Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal.
Maria João Varanda Pereira: Research Centre in Digitalization and Intelligent Robotics (CeDRI), Laboratório para a Sustentabilidade e Tecnologia em Regiões de Montanha (SusTEC), Instituto Politécnico de Bragança, Campus de Santa Apolónia, 5300-253 Bragança, Portugal.

© Daniel Augusto Rodrigues Farina, Rodrigo Campiolo, José Rufino, and Maria João Varanda Pereira; licensed under Creative Commons License CC-BY 4.0
13th Symposium on Languages, Applications and Technologies (SLATE 2024).
Editors: Mário Rodrigues, José Paulo Leal, and Filipe Portela; Article No. 6; pp. 6:1–6:12

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Understanding process-based concurrent programs is challenging for Operating Systems (OS) students. With the aim of facilitating the teaching of this topic, *forkSim* (fork Simulator) was developed, as an educational tool that generates visual representations of the execution flow of such programs in Linux/Unix systems, the typical laboratory environments used in OS classes. In addition to exploring fundamental concepts about the creation of processes and the unpredictability of concurrent programs, the approach undertaken in the development of this work also addresses other topics, such as language processing and program visualization.

Although there are several tools for the simulation and visualization of program execution, many focus on code debugging, aiming to identify errors, as is the case with *GDB* [17] and its GUI front-ends [9]. These debugging tools offer comprehensive interfaces with various features, but they primarily focus on identifying errors and lack visualizations specifically designed for educational purposes. Another option is *Python Tutor* [12, 11, 13], a general-purpose tool to visualize programs written in Python, Java, C, C++, and JavaScript. However, even though *Python Tutor* shows the variables and memory state during program execution, which is essential for teaching programming, it does not support programs with *fork* calls.

Ways to represent *fork* calls may be found on the *Stack Overflow* discussion forums, but these representations are tailored to specific examples, lacking any automation. Furthermore, it is also possible to find tools that can be used to explain concurrent programming, but they focus on thread-based Java programming [7], thus not fitting this work use case.

Considering the constraints and limitations of all these alternatives, it was decided to develop a new tool, specifically tailored to the needs of OS laboratories. This tool goes beyond the visualization of the execution of a conventional (sequential) program and supports the automatic and dynamic visualization of the execution of process-based concurrent programs.

Throughout the development of *forkSim*, several issues arose that needed to be tackled: what information should be obtained during the execution of process-based programs written in C, how to visualize the actions of these programs, and which technologies would be most suitable for transforming and simulating the code. These issues are addressed in Section 2, where the stages of the tool development are explained. Section 3 presents an example of the transformation of the C code, the JSON file resulting from its execution, and the visualization and simulation of the execution steps. Finally, Section 4 presents the conclusions that can be already extracted from the work developed so far, and lays out the next steps.

2 Development

The development of *forkSim* went in two main stages, corresponding to the two main components of this tool. The first stage involved developing a preprocessor of the original C source code, that modifies it to extract relevant information during its execution. The second stage consisted of developing a web interface to visualize the collected actions and the program workflow. These stages and components are next described in detail.

2.1 Code Transformation

To be possible to later visualize the effective execution flow of an instance of a concurrent C program based on the process model, the original source code must be modified such that the relevant actions performed by the program are registered in runtime. In essence, this is an application of *code instrumentation*, a technique commonly used for program visualization [5, 8, 2]. The main idea is to annotate the code with inspection functions (inspectors), making it possible to extract static and dynamic information from the program execution.

To extract the syntax tree from the C language, a Python program was developed using the *PLY* library [3]. Other options were also tried, including the *pycparser* library [4], created specifically for this purpose, and also generic grammar construction tools, such as *Lark* [18], *ANTLR* [16], or *Tree-sitter* [6]. None of these options was adopted, due to their excessive complexity considering the specific needs of *forkSim*. Instead, the C language grammars defined in these tools were used as a basis to develop using *PLY* a simplified version of the C grammar, enough to support the code examples and typical use-cases of the OS classes (all of which were tested and confirmed to behave properly using the simplified grammar).

Listings 1 to 8 show the simplified grammar of the C language implemented using *PLY*. The grammar rules are expressed in the *Backus-Naur Form (BNF)* [1] notation, and include: the initial rules of the parsing process (listing 1), rules about function declarations and function arguments (listing 2), rules for the set of declarations (listing 3), rules for the set of expressions (listing 4), rules for binary expressions (listing 5), rules about variable assignments, references, and literals (listing 6), rules about function calls and control statements (listing 7), and rules about variable declarations (listing 8). The rules marked with "*" are rules that were used to insert inspection functions into the original code.

Listing 1 C language grammar - Part 1 - Begin.

```
*program : program_declarations
program_declarations : program_declarations program_declaration
                     | program_declaration
program_declaration : include
                    | function
include : INCLUDE INCLUDE_PATH
        | INCLUDE STRING
```

Listing 2 C language grammar - Part 2 - Function declaration.

```
function_enter : '{'
*function : INT ID '(' arguments ')' function_enter statements '}'
arguments : arguments ',' argument
          | argument
          | <empty>
argument : INT ID
```

Listing 3 C language grammar - Part 3 - Set of declarations.

```
statements : statements statement
           | statement
statement : expression ';'
          | declaration ';'
          | if_statement
          | while_statement
          | for_statement
          | return_statement
```

Listing 4 C language grammar - Part 4 - Set of expressions.

```
expression : function_call
           | literal
           | reference
           | assign
           | unary_expression
```

```
            | binary_expression
            | post_increment
            | parenthesis
unary_expression : '-' expression
                 | '!' expression
parenthesis : '(' expression ')'
*post_increment : ID PLUS_PLUS
```

Listing 5 C language grammar - Part 5 - Binary expressions.

```
binary_expression : expression '+' expression
                  | expression '-' expression
                  | expression '*' expression
                  | expression '/' expression
                  | expression '%' expression
                  | expression '<' expression
                  | expression '>' expression
                  | expression EQ expression
                  | expression NE expression
                  | expression LTE expression
                  | expression GTE expression
                  | expression AND expression
                  | expression OR expression
```

Listing 6 C language grammar - Part 6 - Assignments, references and literals.

```
assign : ID assign_symbol expression
assign_symbol : '='
              | PLUS_ASSIGN
reference : '&' ID

literal : string
        | integer
        | id
string : STRING
integer : INTEGER
id : ID
```

Listing 7 C language grammar - Part 7 - Function calls and control statements.

```
*function_call : ID '(' ')'
               | ID '(' expression_list ')'
expression_list : expression_list ',' expression
                | expression
*if_statement : IF '(' expression ')' '{' statements '}'
              | IF '(' expression ')' '{' statements '}'
                ELSE '{' statements '}'
*do_while_statement : DO '{' statements '}' WHILE '(' expression ')'
*while_statement : WHILE '(' expression ')' '{' statements '}'
*for_statement : FOR '(' expression ';' expression ';' expression ')'
                '{' statements '}'
*return_statement : RETURN expression ';'
```

Listing 8 C language grammar - Part 8 - Declaration of variables.

```
*declaration : type declaration_list
type : INT
     | ID
declaration_list : declaration_list ',' var
                 | var
var : pointer ID array
pointer : pointer '*'
        | <empty>
array : array '[' INTEGER ']'
      | array '[' ']'
      | <empty>
```

A set of inspection functions was created to obtain data from the actions considered relevant to be visualized. These functions are next listed and described:

- `inspector_function_enter()`: added at the beginning of each function, when processing the `function` grammar rule;
- `inspector_function_exit()`: added to the end of each function, including any premature `return` statements, when processing the `return_statement` and `function` grammar rules when there is no return;
- `inspector_function_call()`: added before invoking a function, when processing the `function_call` and `assign` grammar rule;
- `inspector_variable_declare()`: added after declaring any variable, when processing the `declaration` grammar rule;
- `inspector_variable_assign()`: added after updating any variable, when processing the `assign` and `post_increment` grammar rule;
- `inspector_condition()`: added over any conditional in `if`, `while` and `for` statements, when processing the `if_statement`, `while_statement`, `do_while_statement` and `for_statement` grammar rules;
- `inspector_exit()`: replaces the `exit()` function call, when processing the `function_call` grammar rule;
- `inspector_printf()`: replaces the `printf()` function call, when processing the `function_call` grammar rule;
- `inspector_fork()`: replaces the `fork()` system call, when processing the `function_call` grammar rule;
- `inspector_wait()`: replaces the `wait()` system call, when processing the `function_call` grammar rule;
- `inspector_waitpid()`: replaces the `waitpid()` system call, when processing the `function_call` grammar rule.

Other transformations were applied to the code to integrate the inspection functions. One of them consists of including the inspector library at the beginning of the code during the processing of the `program` grammar rule. Another modification is the adaptation of the `for` statement, replacing it with the use of `while` when processing the `for_statement` grammar rule. This adaptation allows the processing of internal `for` statements, such as counter initialization, loop condition and counter update. Each of these inspection functions is responsible for sending data from these actions to an output file per each running process and for updating the state of the inspector as necessary.

Figure 1 represents the sequence of steps through which the original C source code flows, starting from its instrumentation, and ending in the visualization of its execution.

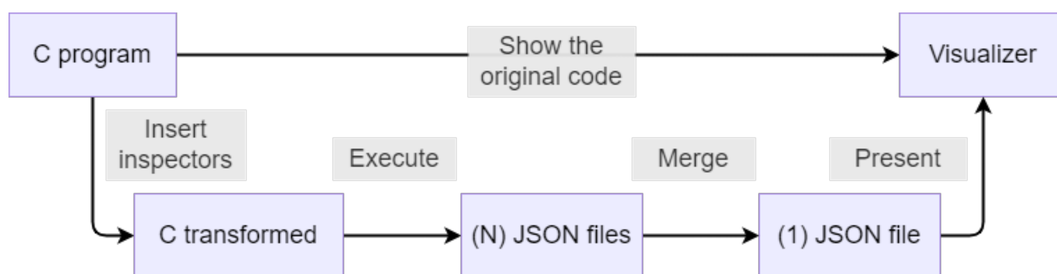

Figure 1 Flow of a concurrent program in *forkSim*: instrumentation, execution and visualization.

The original C code is processed using the *PLY* library to generate an in-memory representation of its structure. This representation is transformed into a new version of the code enriched with inspection functions. During the execution of the instrumented code, the inspectors register the actions performed in JSON files. To avoid race conditions that would occur if all processes shared the same JSON file, there is a distinct temporary output file for each OS process created during the execution of the program. For the purpose of generating the program visualization, the various (N) JSON files are merged into a single file.

Figure 2 shows the TypeScript data type (`ProcessActionFile`) used for the records of the actions registered during the execution of the program. Each record includes: the time in milliseconds at which the action was performed (`time`); the depth in the process tree of the program of the issuing process (`depth`); the process id of the issuing process (`pid`), and the process id of its parent process (`parent_pid`); the source code line number corresponding to the action (`line`); the type of the action being performed (`type`); the name of the function where the action occurred (`function`); and a property to store dynamic values related to the action (e.g., `payload` stores the id of the child process in a *fork* call).

```
type ProcessActionFile = {
  time: number
  depth: number
  pid: number
  parent_pid: number
  function: string
  type: 'fork_enter' | 'fork_exit'
      | 'function_enter' | 'function_exit' | 'function_call'
      | 'condition' | 'printf' | 'wait' | 'waitpid' | 'exit'
      | 'variable_assign' | 'variable_declare'
  line: number
  payload: any
}
```

Figure 2 ProcessActionFile type of each array entry in the JSON file.

On a multi-core machine is possible for several actions in different processes generated by the same program to occur at the same time. This situation is very unlikely in practical terms when using the millisecond time unit. This implies that actions can be aligned sequentially in time even if they are simultaneously, as in single-core systems. The characteristic implicit in Figure 2 is presumed to simplify the generation of combinations of possible actions shown in the web interface. Here, the focus lies on the causal relationship between actions, as in the study of logical clocks in Distributed Systems [15].

2.2 User Interface

The user interface of *forkSim* is a web application that allows to visualize, graphically and interactively, the execution flow of an instance of a process-based program previously executed. In this regard, the web application is not a real-time tracer used to monitor the program during its execution. It actually executes the instrumented version of a program for the first time in order to collect information and then, after this, it works like a simulator, that allows to revisit and visually reproduce the other execution instances with different combinations of flows.

To implement the web application, many tools were tested, but the ones chosen were *React JS* [20], a JavaScript library for creating user interfaces, and *Konva JS* [14], a JavaScript library for creating 2D graphics. These tools were chosen because, in addition to suiting the needs of this project, they are modern and used by a large community of developers.

Figure 3 presents the interface of the web application. The interface is divided into four parts: the code panel (1) on the left, the possible executions panel (2) on the right, the control panel (3, 4, 5) at the bottom, and the visualization section (6, 7, 8) in the center.

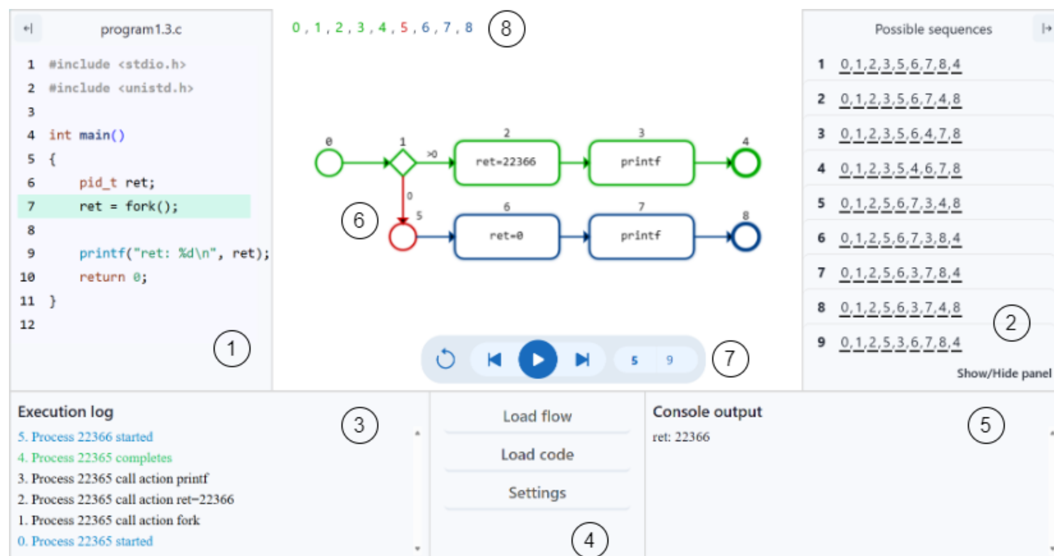

Figure 3 Web application for visualizing the execution of the "program1.3.c" program.

The code panel (1) displays the C source code, highlighting the line of code currently being traced. The possible executions panel (2) presents a selectable list of all possible sequences of actions for the program, assuming the same initial and runtime conditions (e.g., same values for pre-initialized variables and same values for keyboard inputs) as those of the program instance being traced.

The control panel has 3 sections: a log section (3) that shows what was already traced, a configuration section (4), and a console section (5) that shows the program output.

The visualization section shows the execution graph (6) of the particular program instance being simulated, several simulation controls (7), and the particular temporal sequence of actions of that instance (8). The simulation controls (7) make possible to pause/continue the simulation, move forward/backward to a specific action, and execute the simulation in loop. All panels are properly synchronized.

The execution graph was enhanced by the elements of *BPMN diagrams* [10]. *BPMN diagrams* are composed of elements representing activities, decisions, and execution flows, which perfectly fit the purpose of the visualization. The rectangles represent the actions

performed, such as variable declarations, value assignments, and function calls. The diamonds represent the *fork* calls that spawn child processes. The arrows represent the program execution flow. And the circles represent the beginning and end of the processes. The graph representation of the execution that is used in OS classes is illustrated in Figure 4.

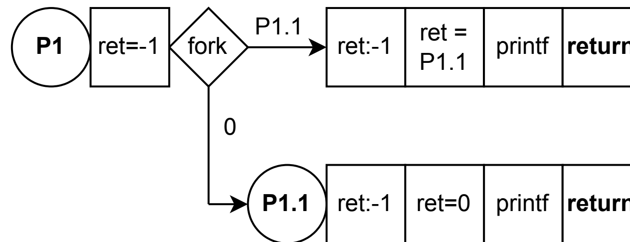

Figure 4 Graph representation used in OS classes for the "program1.3.c" program.

Three different colors were used, to encode the order in time of the various actions: actions yet to be executed (simulated) are represented in blue; those already performed are represented in green; and the current action is represented in red. Note that the diagram is structured in a way that hints at the inherent concurrency/parallelism of the various processes involved: the code paths of a parent process and a direct descendent are represented horizontally side-by-side, to convey this idea. Also, the arrow of time points left to right.

Figure 5 shows the flow from reading the JSON file to the visualization of the program. To create this visualization, a series of procedures are performed. First, the JSON file is read. This file follows the format of an array of `ProcessActionFile`. Then it is converted into an asymmetric matrix (an array of arrays) of `ProcessAction`. Each row of this matrix represents the actions of a specific process. This matrix is essential for interpreting the actions of processes in various operations, such as visualization or the search procedure for possible executions. In the case of visualization, the matrix is converted into an array of `Shape<ProcessAction>`, which contains all the information necessary to create the program visualization using the *Konva JS* library.

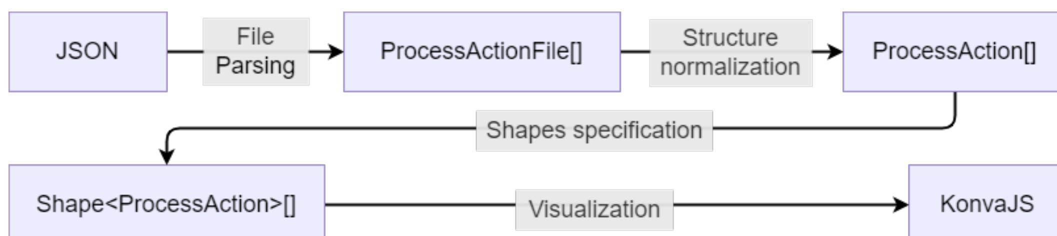

Figure 5 Process flow from the input of the json file, its intermediate transformations and creation of the program view.

Figure 6 illustrates the `ProcessAction` type, a structure that encapsulates the information and actions necessary to interpret the program. A description of each action follows:

- a `ProcessAction {type: "start"}` marks the start of a process, derived from a `ProcessActionFile {type: "function_enter", function: "main"}` or `ProcessActionFile {type: "fork_exit", payload: 0}`;
- a `ProcessAction {type: "fork"}` indicates a fork system call, derived from a `ProcessActionFile {type: "fork_exit", payload: x}`, where x is different of 0;

```
type ProcessAction =
  | { id: number; pid: number; line: number; type: 'start'; payload: null }
  | { id: number; pid: number; line: number; type: 'fork'; payload: null; child_pid: number }
  | { id: number; pid: number; line: number; type: 'action'; payload: { ... } }
  | { id: number; pid: number; line: number; type: 'wait'; payload: { ... } }
  | { id: number; pid: number; line: number; type: 'end'; payload: null }
```

Figure 6 ProcessAction type.

- a `ProcessAction {type: "end"}` marks the end of a process, derived from a `ProcessActionFile {type: "function_exit", function: "main"}` or `ProcessActionFile {type: "exit"}`;
- a `ProcessAction {type: "wait"}` indicates a wait or waitpid system call, derived from a `ProcessActionFile {type: "wait" | "waitpid"}`;
- a `ProcessAction {type: "action"}` is a generic representation of an action, derived from a `ProcessActionFile {type: "function_call" | "printf" | "condition" | "variable_assign" | "variable_declare"}`.

It is worth mentioning that `ProcessActionFile {type: "fork_enter"}` does not have a corresponding `ProcessAction`, as its information is not essential for the interpretation of the program. This arrangement of information facilitates the understanding of how the program works and the implementation of operations in the application.

3 Practical Example

Using the code in Listing 9 as an example, transformation processing is applied to achieve the result shown in Listing 10. After running this modified program and processing its result, the JSON file illustrated in Figure 7 is obtained.

Listing 9 Example code.

```c
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
    pid_t ret;
    ret = fork();
    if (ret == 0) {
        exit(0);
    }
    waitpid(ret, NULL, 0);
    printf("PARENT %d of the CHILD %d\n", getpid(), ret);
    getchar();
    return 0;
}
```

Listing 10 Example code transformed.

```c
#define INSPECTOR_IMPLEMENTATION
#include <inspector.h>
#include <stdio.h>
```

```
#include <sys/wait.h>
#include <unistd.h>
#include <stdlib.h>
int main() {
    inspector_function_enter("main", 5);
    pid_t ret;
    inspector_variable_declare("main", 5, "pid_t", "ret");
    ret = inspector_fork("main", 6);
    inspector_variable_assign("main", 6, "pid_t", "ret", &ret);
    if (inspector_condition("main", 7, "if", ret == 0, "ret == 0")) {
        inspector_exit("main", 8, 0);
    }
    inspector_waitpid("main", 10, ret, NULL, 0);
    inspector_printf("main", 11, "PARENT %d of the CHILD %d\n",
                        getpid(), ret);
    inspector_function_call("main", 12, "getchar");
    getchar();
    inspector_function_exit("main", 13);
    return 0;
}
```

During processing, the inspector library header is added to the program. Furthermore, the `inspector_function_enter()` and `inspector_function_exit()` functions are inserted at the beginning and end of all program functions. Some functions, such as `printf()` and `fork()`, are literally intercepted to be treated individually during execution, taking the form `inspector_[original name]()`. Other functions are simply marked with the `inspector_function_call()` function call. In the case of conditionals, the conditional expression is passed literally and as a string to the `inspector_condition()` function, allowing the processing of the expression and its result. After being processed and transformed, the code takes on the format shown in Listing 10.

After this code is compiled and executed, a set of files are created in JSON format. After the program finishes, these files are merged into a single JSON file – see Figure 7.

Figure 8 presents the visualization of the program "example.c" in the web application. The visualization build process comprises reading the JSON with the program execution trace, transforming the JSON into intermediate representations, and finally rendering the visualization using the *Konva JS* library.

4 Conclusion

Concurrent and system-level programming is typically regarded as an hard topic by students of Operating Systems. In turn, teachers may also find challenging to convey and illustrate the concepts involved, in a way that makes the teaching/learning process more effective.

forkSim is a tool that offers an automated solution for this problem, integrating concepts of concurrent programming, visualization interfaces, automatic code inspection with language processors, programming language grammars, and JSON-based data representation. The tool allows teachers to illustrate the behavior of process-based concurrent programs without manually drawing their execution flow, and helps students to study and comprehend them.

However, it is important to acknowledge the limitations of *forkSim*. User experience may degrade for programs with a high volume of actions performed by the program, potentially impacting usability. Additionally, specific C features might not translate perfectly, affecting visualization accuracy in certain cases.

```
[
  { "time": 1694740659831057,  "depth": 0,  "pid": 434, "parent_pid": 146, "function": "main", "line": 5
  , "type": "function_enter",  "payload": "" },
  { "time": 1694740659832111,  "depth": 0,  "pid": 434, "parent_pid": 146, "function": "main", "line": 5
  , "type": "variable_declare","payload": { "type": "pid_t", "name": "ret" } },
  { "time": 1694740659832439,  "depth": 0,  "pid": 434, "parent_pid": 146, "function": "main", "line": 6
  , "type": "fork_enter",      "payload": "" },
  { "time": 1694740659832744,  "depth": 0,  "pid": 434, "parent_pid": 146, "function": "main", "line": 6
  , "type": "fork_exit",       "payload": 435 },
  { "time": 1694740659833038,  "depth": 0,  "pid": 434, "parent_pid": 146, "function": "main", "line": 6
  , "type": "variable_assign", "payload": { "type": "pid_t", "name": "ret", "value": 435 } },
  { "time": 1694740659833739,  "depth": 0,  "pid": 434, "parent_pid": 146, "function": "main", "line": 7
  , "type": "condition",       "payload": { "cause": "if", "value": 0, "expression": "ret == 0" } },
  { "time": 1694740659838276,  "depth": 0,  "pid": 434, "parent_pid": 146, "function": "main", "line": 10
  , "type": "waitpid",         "payload": { "pid": 435, "status": 0 } },
  { "time": 1694740659838574,  "depth": 0,  "pid": 434, "parent_pid": 146, "function": "main", "line": 11
  , "type": "printf",          "payload": "PARENT 434 of the CHILD 435\n" },
  { "time": 1694740659838829,  "depth": 0,  "pid": 434, "parent_pid": 146, "function": "main", "line": 12
  , "type": "function_call",   "payload": { "function": "getchar" } },
  { "time": 1694740661511736,  "depth": 0,  "pid": 434, "parent_pid": 146, "function": "main", "line": 13
  , "type": "function_exit",   "payload": "" },
  { "time": 1694740659836051,  "depth": 1,  "pid": 435, "parent_pid": 434, "function": "main", "line": 6
  , "type": "fork_exit",       "payload": 0 },
  { "time": 1694740659836921,  "depth": 1,  "pid": 435, "parent_pid": 434, "function": "main", "line": 6
  , "type": "variable_assign", "payload": { "type": "pid_t", "name": "ret", "value": 0 } },
  { "time": 1694740659837161,  "depth": 1,  "pid": 435, "parent_pid": 434, "function": "main", "line": 7
  , "type": "condition",       "payload": { "cause": "if", "value": 1, "expression": "ret == 0" } },
  { "time": 1694740659837377,  "depth": 1,  "pid": 435, "parent_pid": 434, "function": "main", "line": 8
  , "type": "exit",            "payload": "" }
]
```


Figure 7 JSON result of the execution of the transformed code.

Figure 8 Visualization of the "example.c" program.

Despite such limitations, *forkSim* offers several key advantages. Designed primarily for educational use, it allows users to explore the behavior of concurrent C programs, fostering a deeper understanding of these concepts. Evaluating *forkSim* using the guidelines of a Learning Object [19] and conducting experiments in the classroom can bring new ideas and functionalities to enhance its effectiveness in achieving its educational objectives.

References

1 Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 2006.

2 Carlos Balsa, Luís M. Alves, Maria João Pereira, Pedro Rodrigues, and Rui Lopes. Graphical simulation of numerical algorithms : an approach based on code instrumentation and java technologies. *CSEDU 2012 - 4th International Conference on Computer Supported Education*, 2012. URL: `http://hdl.handle.net/10198/6998`.

3 David Beazley. Library ply. `https://github.com/dabeaz/ply`. Accessed: 2024-03-12.

4 Eli Bendersky. Library pycparser. `https://github.com/eliben/pycparser`. Accessed: 2024-03-12.

5 Mário Berón, Pedro Henriques, Maria João Pereira, and Roberto Uzal. Static and dynamic strategies to understand c programs by code annotation. In *1st International Workshop on Fondations and Techniques for Open Source Software Certification*, Braga, 2007.

6 Max Brunsfeld. Library tree-sitter. `https://github.com/tree-sitter/tree-sitter`. Accessed: 2024-03-12.

7 Max Brunsfeld. Transpotting problem in laboratory class using tsim software. `https://github.com/tree-sitter/tree-sitter`. Accessed: 2024-03-12.

8 Daniela Cruz, Mário Berón, Pedro Henriques, and Maria João Pereira. Code inspection approaches for program visualization. *Acta Electrotechnica et Informatica*, 9(2):32–42, 2009.

9 Free Software Foundation. Gdb front ends and other tools. `https://sourceware.org/gdb/wiki/GDB%20Front%20Ends/`. Accessed: 2024-02-04.

10 Object Management Group. Bpmn specification - business process model and notation. `https://www.bpmn.org/`. Accessed: 2024-02-29.

11 Philip Guo. Ten million users and ten years later: Python tutor's design guidelines for building scalable and sustainable research software in academia. In *The 34th Annual ACM Symposium on User Interface Software and Technology*, UIST '21, pages 1235–1251, New York, NY, USA, 2021. Association for Computing Machinery. `doi:10.1145/3472749.3474819`.

12 Philip J. Guo. React. `https://pythontutor.com/`. Accessed: 2024-02-19.

13 Philip J. Guo. Online python tutor: embeddable web-based program visualization for cs education. In *Proceeding of the 44th ACM Technical Symposium on Computer Science Education*, SIGCSE '13, pages 579–584, New York, NY, USA, 2013. Association for Computing Machinery. `doi:10.1145/2445196.2445368`.

14 KonvaJS. Konvajs. `https://konvajs.org/`. Accessed: 2024-02-19.

15 Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. `doi:10.1145/359545.359563`.

16 Terence Parr. Library antlr. `https://github.com/antlr/antlr4`. Accessed: 2024-03-12.

17 GNU Project. Gdb: The gnu project debugger. `https://sourceware.org/gdb/`. Accessed: 2024-02-04.

18 Erez Shinan. Library lark. `https://github.com/lark-parser/lark`. Accessed: 2024-03-12.

19 Milene Selbach Silveira and Mára Lúcia Fernandes Carneiro. Diretrizes para a avaliação da usabilidade de objetos de aprendizagem. In *Simpósio Brasileiro de Informática na Educação*, Rio de Janeiro, November 2012.

20 Meta Open Source. React. `https://pt-br.react.dev/`. Accessed: 2024-02-19.