Visualizing Execution Traces in RISCAL*

Wolfgang Schreiner
Research Institute for Symbolic Computation (RISC)
Johannes Kepler University, Linz, Austria
Wolfgang.Schreiner@risc.jku.at

William Steingartner
Department of Computers and Informatics
Faculty of Electrical Engineering and Informatics
Technical University of Košice, Slovakia
William.Steingartner@tuke.sk

March 28, 2018

Abstract

We report on initial results concerning the visualization of execution traces of algorithms that are formally specified and modeled in the RISC Algorithm Language (RISCAL). These algorithms are executed and visualized in the associated software system which also validates their correctness by checking the satisfaction of the formal contracts. This work has been stimulated by corresponding visualization of Jane, a language with an associated toolkit that has been developed to demonstrate the categorical semantics of programming languages. By the new visualization extension of RISCAL, the suitability of the software for the purpose of computer science education shall be improved.

1 Introduction

The RISC Algorithm Language (RISCAL) [2, 3] is is a specification language and associated software system for modeling mathematical algorithms, formally specifying their behavior based on mathematical theories, and validating the correctness of algorithms, specifications, and theories by execution/evaluation. The goal of the system is to make it easy to find errors in algorithms (abstract programs), their formal specifications (pre- and postconditions) and their formal annotations (loop invariants, termination measures) before actually attempting to prove those verification conditions that are derived from the annotated program in such a way that their validity implies the correctness of the program with respect to its specification. Observation shows that much time is wasted in proof attempts that are a priori doomed to fail because of errors in programs, specifications, and annotations. However, a failed proof attempt (even if performed by an automated reasoner or with the help of an interactive proof assistant) does not indicate the reason of a failure: an inadequacy in the program, in the specification, in the annotations, or simply a failure to find an adequate proof strategy.

Therefore RISCAL pursues the strategy to "fully automatically" check programs and annotations. For this purpose, it restricts the domains of all types by formal parameters to finite sizes; e.g., the type `Array[N,`

*Supported by the Austrian OEAD WTZ program and the Slovak SRDA agency under the contract SK 14/2018 "SemTech — Semantic Technologies for Computer Science Education" and by the Johannes Kepler University Linz, Linz Institute of Technology (LIT), Project LOGTECHEDU "Logic Technology for Computer Science Education".

$\mathbb{Z}$`[-M,M]` denotes the type of all arrays of size N whose elements are integers in the interval $[-M, M]$. All program variables and all logical variables are confined to such types such that it is possible on the one hand to check all possible executions of a program and on the other hand to decide all formulas. To this end, RISCAL translates, for user-provided values of the domain bounds, programs and formulas to executable versions of their (generally non-deterministic) denotational semantics which are invoked for the execution of programs and the evaluation of formulas. Thus it is possible to automatically check the correctness of a program for particular instances of the class of its possible models. If it is not possible to falsify the correctness for these instances, this validates (i.e., increases the confidence in) the general correctness for the whole class; the general correctness may be subsequently established by proof-based verification.

So far, the information provided by RISCAL was based on textual information; every check of a procedure application displayed the arguments of the procedure, its results, and information whether all specifications and annotations were satisfied. To get more information from a procedure, it can be annotated by `print` commands that display intermediate values of the variables modified in the procedure; furthermore, expressions in predicates can be annotated as `print` expressions that as a side-effect of their evaluation print their values. However, this is a rather clumsy way of investigating the dynamic behavior of the execution of procedures respectively the evaluation of formulas.

A more elegant approach is to visualize the execution of formal models by graphical means. There exist various approaches to this problem; e.g. [1] describes how a controller formally specified in the B method is integrated with a computer game engine to visualize in a virtual environment the effects of the controller. Another approach has been taken in a recent toolset for the toy programming language Jane [7, 5] that has been developed for education on the formal semantics of programming languages. This toolset illustrates the categorical semantics of Jane [6] as a directed graph whose nodes (the objects of the category) represent the states of a procedure execution and whose arrows (the morphisms of the category) represent the transitions among the procedure states. Starting with the node representing the initial procedure state and following the arrows representing the state transitions, the program execution can be understood.

Furthermore, this representation is hierarchical: a procedure call is modeled by a functor from the category of the caller's state space to the category of the callee's state space; a procedure return is modeled by a functor in the reverse direction. A program execution can be thus visualized as a directed graph whose nodes are categories (state spaces of procedure invocations) linked by functors (procedure calls and returns); each category is again represented by a graph whose nodes are category objects (procedure states) linked by morphisms (state transitions).

The work presented in this paper on a RISCAL visualization of procedure executions has been inspired by this visualization of Jane; it also yields a hierarchical graph of graphs where each graph represents the state changes in the execution of a procedure; the user may switch by mouse-clicks from a graph on one level to a graph on the next level and back.

The rest of the paper is organized as follows: Section 2 demonstrates the features of the visualization and its practical use; Section 3 discusses the implementation of the visualization and the underlying technology; Section 4 concludes and presents an outlook on future work.

2 Use of the Visualization

If RISCAL is started and trace visualization is enabled (see Section 3), the graphical user interface depicts a check box "Trace" (see Figure 1). If this option is selected, every run/check of a RISCAL operation (procedure, function, predicate, theorem) opens a window in which an execution trace of this operation is displayed *provided that*

- the option "Nondeterminism" is *not* selected,
- the options "Multi-Threaded" and "Distributed" are *not* selected.

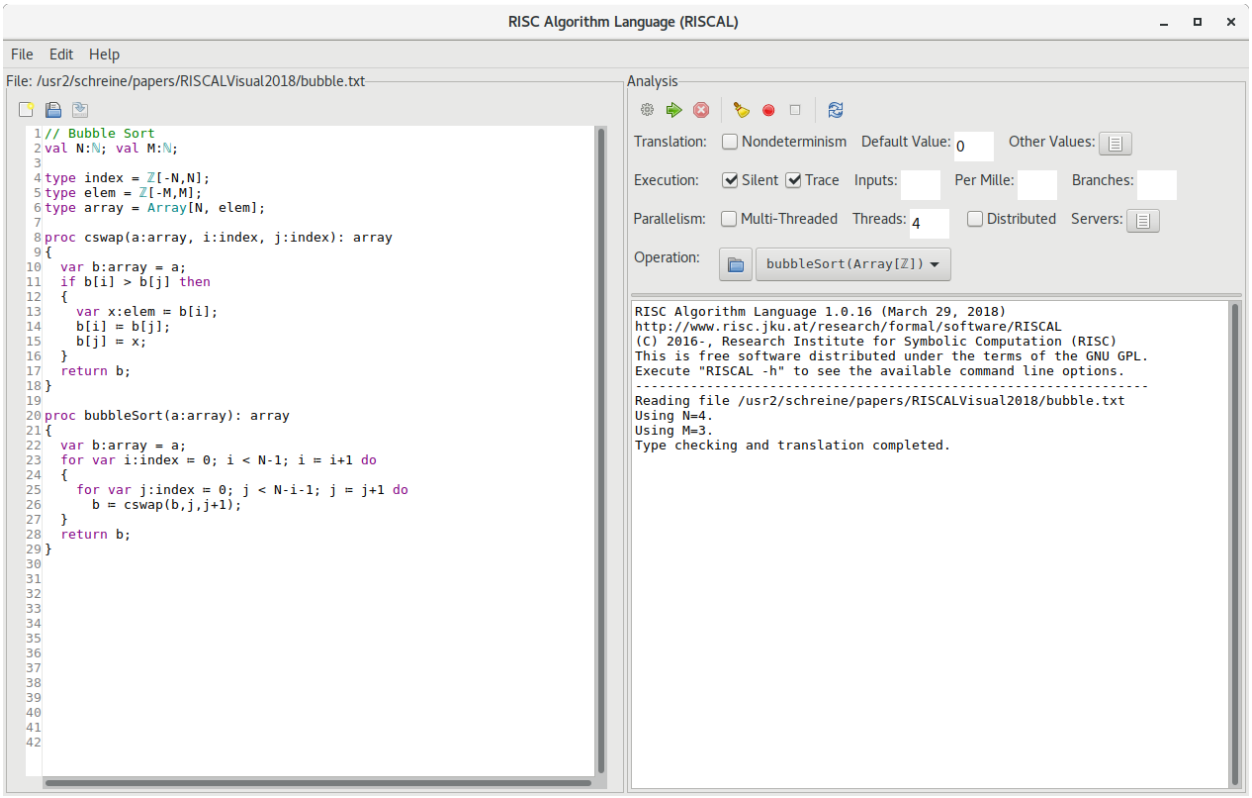

Figure 1: The RISCAL User Interface

In other words, visualization is restricted to deterministic execution of RISCAL operations in the sequential mode of the RISCAL software.

Furthermore, the visualization essentially only displays the changes of variables by the commands in the body of a procedure; for functions (including predicates and theorems) whose result is determined by the evaluation of an expression the computation of the result is displayed as a single step (except that also calls of other operations are displayed). The visualization is thus currently mainly useful for understanding the operational behavior of a procedure.

We demonstrate this by a visualization of the following RISCAL model:

```
// Bubble Sort
val N:ℕ; val M:ℕ;

type index = ℤ[-N,N];
type elem = ℤ[-M,M];
type array = Array[N, elem];

proc cswap(a:array, i:index, j:index): array
{
  var b:array = a;
  if b[i] > b[j] then
  {
    var x:elem := b[i];
    b[i] := b[j];
    b[j] := x;
  }
  return b;
```

```
}

proc bubbleSort(a:array): array
{
  var b:array = a;
  for var i:index := 0; i < N-1; i := i+1 do
  {
    for var j:index := 0; j < N-i-1; j := j+1 do
      b := cswap(b,j,j+1);
  }
  return b;
}
```

This model implements a version of the "Bubble sort" algorithm as a procedure `bubbleSort` that uses an auxiliary procedure `cswap` for the conditional swap of two array elements.

If we select the operation `bubbleSort` and set with the button "Other Values" the parameters N and M to 4 and 3, respectively, the first check of the procedure is performed with the argument array $a = [-3, -3, -3, -3]$. Since here actually no swap is performed, we close the window that pops up immediately (by clicking on the top-right button in the window frame). This lets the execution proceed to the second check with $a = [-2, -3, -3, -3]$ whose execution is displayed by another window; it is this window that is shown on the top of Figure 2.

The window displays in the title the operation invocation `bubbleSort([-2,-3,-3,-3])` whose execution is visualized; the tag "Level 0" indicates that this is the top-level of the execution (every entry into the visualization of a nested procedure call increases this level by one). In the window, we see a directed graph (a linear sequence of nodes) that are connected by directed edges (arrows) and that are laid out in a two-dimensional manner from left to right and top to bottom. This graph has three kinds of nodes:

- *Numbered nodes* represent the sequence of states constructed by performing assignments to the variables of the procedure; such a node is always the target of a solid arrow (see below). By hovering with the mouse pointer over such a node, a small window pops up that displays the values of the various variables in that state (see the node numbered 17 in the figure).

- *Empty nodes* represent "intermediate" states that have the same variable values as the previous state; such a node is always the target of a dashed arrow (see below). An empty node is displayed separately, because it indicates an event that helps to understand how the state indicated by the next numbered node has been computed (see the explanation of the dashed arrows below).

- *Call nodes* display a small window with a graph symbol; such a node represents the call of another operation, the header of this window displays the call itself. A graph node is always the target of a dashed arrow (see the explanations below).

Nodes may be selected by a mouse click and moved to another location in the display.

Nodes are connected by two types of arrows:

- *Solid arrows* (which lead to a numbered node) represent changes from one state to another by the modification of a variable (thus generally some variable value in the target node is different from the value of the corresponding variable in the source node). The label associated to the arrow displays the name of the command that is responsible for the change and in the second line (within parentheses) the new value of the variable. State changing commands are variable assignments but also various kinds of `choose` statements that nondeterministically set variables to values satisfying given conditions.

- *Dashed arrows* (which lead to an empty node) represent events within the change from one state to the next that contribute to the change but do not change the state themselves. We consider as such events on

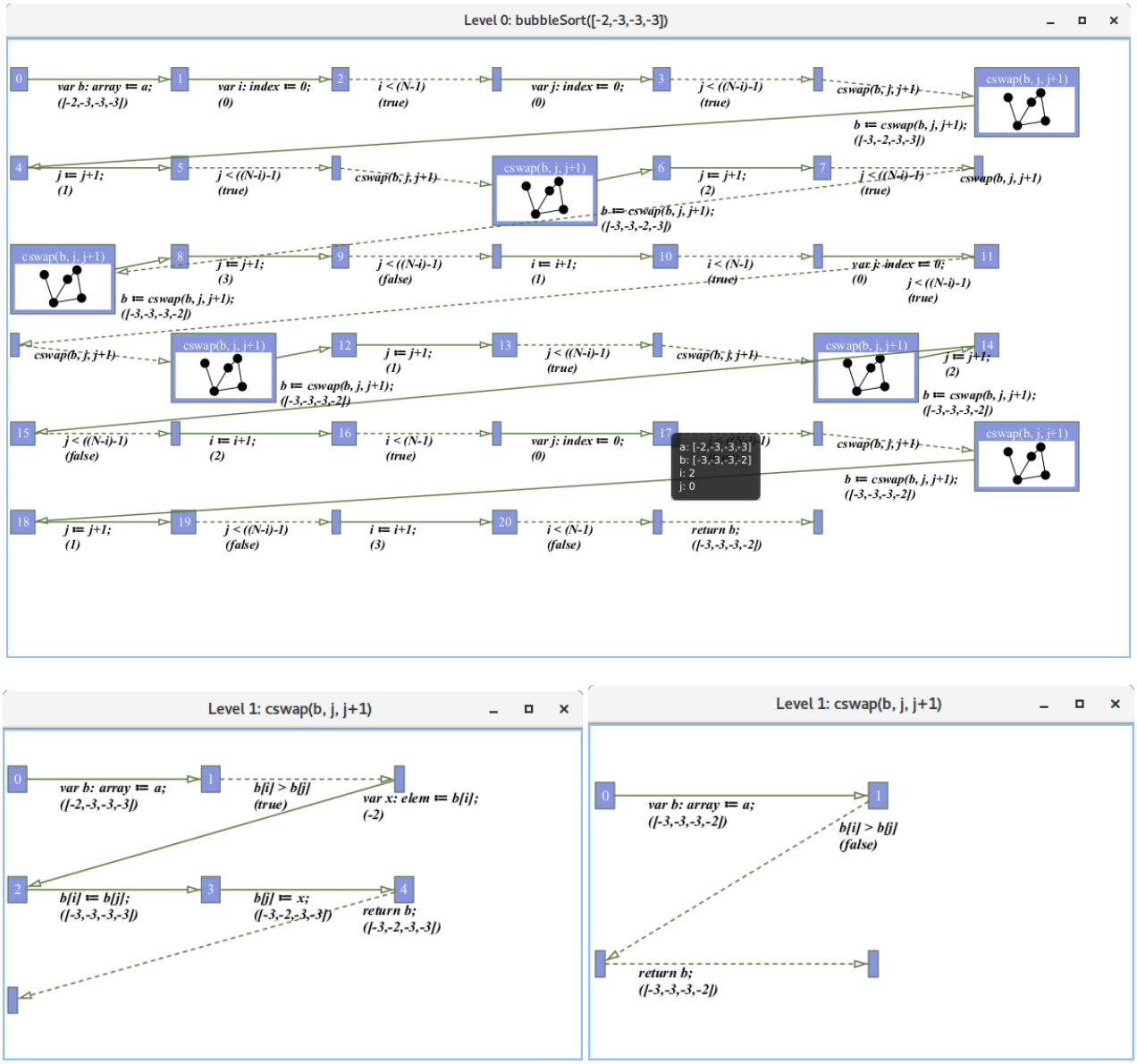

Figure 2: Bubble Sort and Conditional Swap

the one hand the testing of boolean conditions (respectively the matching of recursive structures against patterns) that direct the control flow of statements (conditionals and loops) and on the other hand the application of an operation (procedure or function) whose execution yields a value. The label associated to the arrow displays the test expression respectively the application expression in the first line and the result of the test respectively of the application (within parentheses) in the second line.

The labels associated to the arrows (actually to the source nodes of the arrows) may be selected by a mouse click and moved to another location in the display. In fact, some of the arrow labels displayed in the figure have been manually moved for greater clarity.

By double-clicking on a call node which represents the application of an operation the content of the window is modified to visualize the execution of that operation. The left diagram at the bottom of Figure 2 displays the execution corresponding to one application of `cswap(b,j,j+1)` where the test determines that the elements at positions j and $j+1$ are not in the right order such that it is necessary to swap the two elements by three assignments; the right diagram displays an application where the test determines that the elements are already in the right order such that no swap is necessary. The tag "Level 1" in the titles of the windows indicates that the visualization refers to the application of an operation that was invoked on "Level 0". If the "Level 1" operation would involve another operation application, it would contain another call node; by double-clicking on that node, we would move to "Level 2" and so on. A double click on any empty part of the window moves the display back to the previous level.

By closing the window (via a click on the top right button of the window frame) the visualization is terminated and the execution machine continues with the execution of the next invocation of the operation being checked. By clicking on the red "Stop Execution" button in the main window (see Figure 1), this process can be prematurely terminated (such that it is not necessary to step through the visualizations of all possible calls of the operation).

The visualization of the execution trace of a procedure also displays the application of all operations (typically functions and predicates) applied when checking the formal annotations (pre- and postconditions, invariants, termination measures, assertions) of the procedure. If this is not desired, those annotations should be temporarily commented out.

3 Implementation of the Visualization

Due to dependencies on external software (see below), the execution trace visualization of RISCAL is only enabled, if the command line option `-trace` is provided (this is hidden from the user: the local installation of the `RISCAL` script does or does not set this option).

The visualization has been implemented with the help of the Eclipse GEF/Zest framework for graph visualization[1]. This software depends on the graphics framework JavaFX[2] which is part of the Oracle JDK 9 implementation of Java; the use of this version of Java is thus strongly recommended, if the visualization feature is to be used (otherwise also any version of JDK 8 is fine).

The Java open source implementation OpenJDK 9 does not yet support JavaFX. If for some reason Oracle JDK 9 cannot be used (e.g., Oracle JDK 9 is not available for 32 bit operating systems) but the visualization is nevertheless desired, the local installation must provide OpenJDK 8 and the corresponding OpenJFX 8 implementation of JavaFX. However, then the Eclipse SWT toolkit on which RISCAL is based must not use the current version GTK3 of the graphical toolkit GTK but the older version GTK2 (which gives RISCAL a somewhat outdated look and feel). This can be achieved by setting the environment variable `SWT_GTK3` to `0`.

The `README` file of the distribution contains more detailed information on the configuration of RISCAL for the use of the trace visualization extension. The RISCAL site [2] provides a 32 bit virtual machine with RISCAL preinstalled; in that machine the command `RISCAL-trace` invokes RISCAL with trace visualization enabled but only using GTK2; the command `RISCAL` uses GTK3 but has the visualization disabled.

[1] https://github.com/eclipse/gef/wiki/Zest
[2] https://docs.oracle.com/javafx/2/overview/jfxpub-overview.htm

4 Conclusions

The visualization of the sequence of the individual state-changing statements executed in the body of a procedure (accompanied by the visualization of the decisions that led to the selection of these statements and the visualization of the operations called within these statements) may significantly contribute to the understanding the dynamical behavior of a procedure. We will investigate the actual usefulness of such visualizations by the application of RISCAL in future courses for students of computer science and mathematics.

Future work will also deal with the complementary visualization of the evaluation of expressions (terms and formulas), which, e.g., denote the results of non-state changing operations (functions and predicates) but are also at the the core of a procedure's specification and its annotations. Especially the visualization of the evaluation of quantified expressions (in particular universally and existentially quantified formulas) may prove helpful in understanding why certain conditions hold respectively do not hold. Thus not only the behavior of a procedure but also the reasoning about its correctness properties may become more transparent.

References

[1] Štefan Korečko and Branislav Sobota. "Computer Games as Virtual Environments for Safety-Critical Software Validation". In: *Journal of Information and Organizational Sciences* 41.2 (2017), pp. 197–212. URL: https://jios.foi.hr/index.php/jios/article/view/1064.

[2] RISCAL. *The RISC Algorithm Language (RISCAL)*. Mar. 2017. URL: https://www.risc.jku.at/research/formal/software/RISCAL.

[3] Wolfgang Schreiner. *The RISC Algorithm Language (RISCAL) — Tutorial and Reference Manual (Version 1.0)*. Technical Report. Download from [2]. Johannes Kepler University, Linz, Austria: RISC, Mar. 2017.

[4] Wolfgang Schreiner, Alexander Brunhuemer, and Christoph Fürst. "Teaching the Formalization of Mathematical Theories and Algorithms via the Automatic Checking of Finite Models". In: *Post-Proceedings ThEdu'17, 6th International Workshop on Theorem proving components for Educational software*. Ed. by Pedro Quaresma and Walther Neuper. Vol. 267. Electronic Proceedings in Theoretical Computer Science (EPTCS). Gothenburg, Sweden, August 6, 2017: Open Publishing Association, 2018, pp. 120–139. URL: https://doi.org/10.4204/EPTCS.267.8.

[5] William Steingartner, Mohamed Ali M. Eldojali, Davorka Radakovic, and Jiří Dostál. "Software support for course in Semantics of programming languages". In: *IEEE 14th International Scientific Conference on Informatics*. Poprad, Slovakia, November 14–16, 2017, pp. 359–364. URL: https://www.researchgate.net/publication/321341571_Software_support_for_course_in_Semantics_of_programming_languages.

[6] William Steingartner and Valerie Novitzká. "Categorical Semantics of Programming Langages". In: *Selected Topics in Contemporary Mathematical Modeling*. Ed. by Andrzej Z. Grzybowski. Vol. 331. Monographs. Czestochowa University of Technology. Chap. 11, pp. 167–192. URL: https://doi.org/10.17512/STiCMM2017.11.

[7] William Steingartner and Valerie Novitzká. "Learning tools in course on semantics of programming languages". In: *MMFT 2017 — Mathematical Modelling in Physics and Engineering*. Czestochowa, Poland, September 18–21, 2017, pp. 137–142. URL: http://im.pcz.pl/konferencja/get.php?doc=MMFT2017_streszczenia_wykladow.pdf.