Visualizing Permission Flow of Concurrent Programs

J. J. Brinkman
University of Twente
P.O. Box 217, 7500AE Enschede
The Netherlands
j.j.brinkman@student.utwente.nl

ABSTRACT

The output of concurrent specification and verification tools is difficult to understand for the average programmer. In this paper we discuss the development of a visualization tool that visualizes the permission flow of concurrent programs. This tool might help programmers to better understand the output given by their verification tools and the flow of permissions in their concurrent programs. We implemented a prototype of this visualization tool using VerCors and Prefuse. The prototype uses delta calculations to visualize permission changes over the execution.

Keywords

Concurrency, Visualization, VerCors, Verification, Permission flow, Prefuse, Debugging, Permissions.

1. INTRODUCTION

It is not uncommon for bugs to be accidentally introduced during software development. Because of this it is important to verify the behavior of the program. Many methods of program verification have been developed e.g. unit tests, acceptance tests, etc. Alternatively, analyzing programs can be done using static verification or runtime verification. Program verification is a formal method that can prove whether the program satisfies a formal specification of its behavior. Assuming the specification is correct and complete, a correct program should not contain any bugs.

```
//Simple counter with 2 threads
class{
    int c = 0;
    void run(){
        parallel (tid= 0..2){
            for(int i=0;i<5;i++){
                c=c+1;
            }
        }
    }
}
```

Figure 1. Example program in PVL.

Concurrent programs can have hard-to-find bugs such as bugs that could crash the entire program by creating data races or deadlocks. Because of this, various static verification tools have been developed for concurrent programs such as VerCors [2] and Chalice [5]. Stijn Gijsen has extended VerCors with a runtime permission checker [4]. VerCors uses a native programming and annotation language (PVL).

The example program in figure 1 is an example of an incorrect concurrent program. This program contains a data race on the integer *c*. The program in Figure 2 avoids this data race by specifying the permissions on the integer *c* using an invariant and an atomic statement. The invariant proves the correctness of the program and specifies the scope of the atomic, whereas the atomic prevents the actual data race from happening.

```
//Simple counter with 2 threads
class{
    int c = 0;

    requires perm(c,1)
    void run(){
        invariant inv(perm(c,1)){
            parallel (tid= 0..2){
                for(int i=0;i<5;i++){
                    atomic(inv){
                        c=c+1;
                    }
                }
            }
        }
    }
}
```

Figure 2. Example program without data race.

The output of a verification results in either a "pass" or a "fail" with an error message, which specifies which assertion on which line of the program failed the verification. The VerCors verifier outputs its results on the commandline.

VerCors specifications are relatively complex because they have to account for memory accesses and keep track of the various read and write permissions of all threads. Because of this high complexity it is common to introduce bugs or inconsistencies in the specification of the program. Just looking at the simple example in Figures 1 and 2 the correct program added 6 additional lines to the original, which is half the original program. If a program is proven correct using a faulty specification it does not contain any dataraces but the program might still contain deadlocks or other unwanted behaviour. For this reason it is important that programmers understand the proof or, when a proof could not be generated, why the verification failed. And when necessary they should have the possibilty of checking the permission flow manually.

Since the proofs and results of the VerCors verification are complex and rather difficult to understand without additional knowledge about the verification method used, a visualization tool could be very helpful for programmers that would like to understand this verification. To help the programmer understand the verification and the proof generated by VerCors we will develop a visualization tool that can visualize the permission flows in a concurrent program.

1.1 Problem Statement

Even though VerCors can use static or runtime verification to prove a program correct or incorrect according to its specification, it is often difficult to understand this proof without deeper knowledge of the inner workings and model of VerCors. This makes it difficult for the programmer to understand whether his specification is correct or not.

In order to make it easier for the average programmer to understand the VerCors verification some kind of visualization tool is needed. This tool should visualize all permissions of the threads in the program and how they are exchanged between threads during execution of the program, also known as the permission flow. An important part of this visualization tool is visualizing the permissions in a clear way.

However, there is also a challenge in visualizing more complex programs with more data, larger arrays and/or larger amounts of threads. Since methods for visualizing smaller amounts of permissions might not be clear enough when the amount of permissions grows we will have to find a way to effectively visualize both large and small amounts of permissions.

1.2 Research Questions

Based on the problem statement there is one main research question which should be addressed.

How can the permission flow of a concurrent program be visualized?

This research question can be split into three sub questions.

How can the permissions of a thread at a given point in the execution be visualized?

How can the changes in permissions of a thread be visualized when stepping to the next point in the execution?

How can large amounts of threads and/or more moving data be visualized?

2. BACKGROUND

Permissions in the VerCors verification model are defined as rational numbers in the range 0 to 1. Within this range a permission value of 1 equals a writing permission and 0 means there is no permission at all. However to allow for easy exchange of read permissions 0^+ has been added to this range of numbers. 1^- has also been added as a counterpart to 0^+. Important to note is the fact that both 0^+ and 1^- are read permissions. In this context 0^+ means a negligible amount more than 0, similarly 1^- is slightly less than 1.

3. RELATED WORK

There exists related work on visualization methods and standards.

Shneideman discusses a visualization mantra, GUI design and how to visualize data [7]. “A useful starting point for designing graphical user interfaces is the Visual Information-Seeking Mantra: overview first, zoom and filter, then details on demand.” We used this mantra for the GUI design of the tool and of course for the tool itself.

Carpendale discusses visualization and how to distinguish data and/or information using visual variables [3]. The discussion in this report centers on how the visual variables can be used in the creation of visual representations for the purpose of information visualization. We use the visual variables for distinguishing between outgoing and incomming permissions. Additionally, visual variables were used to differentiate between read, readwrite and no permission at all.

4. APPROACH

The VerCors model is already capable of parsing programs and verifying them. To get access to the verification data Stefan Blom has implemented a dynamic interpreter for this verification. This interpreter is initialised with a textfile that contains the program. Once initialised it can be queried for possible steps, this allows the visualization tool more control in the interleaving of various threads. The tool could for example give the user control over the interleaving using this functionality. If there are no possible next steps the program is either deadlocked or finished. An example of the interaction with the interpreter can be found in figure 3.

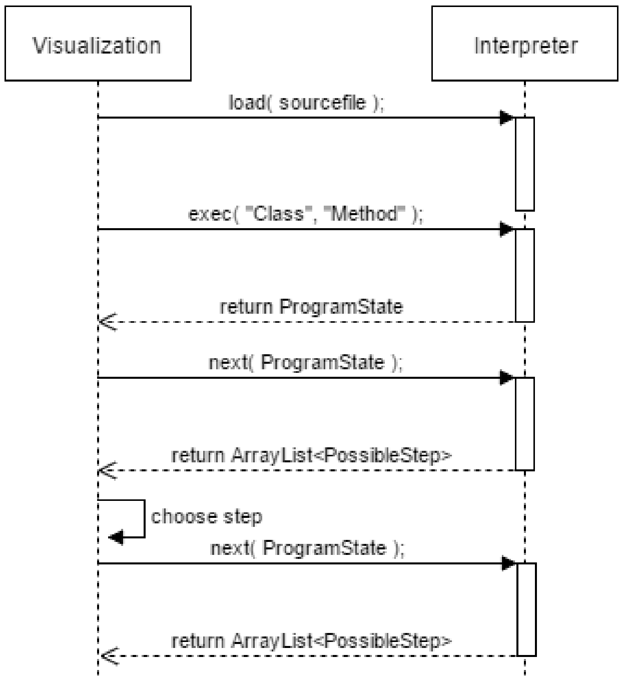

Figure 3. Communication between the visualization and the interpreter.

However, to make communication about the state of the program possible between the interpreter and the visualization tool we had to develop a datastructure called the `ProgramState`. The `ProgramState` contains an abstraction of the heap, an abstraction of the threads, an abstraction of all locals of those threads and most importantly it contains the resources those threads possess.

Using a `PossibleStep` object it is possible to query the interpreter for the `ProgramState` that corresponds to the next step. The visualization tool uses the `ProgramState` to create a simplified model of the permissions that are present in the program. A single permission within this model is the permission a single thread has on a single object on the heap in a rational representation. The permission model uses the resources a thread possesses to derive the rational value for a given permission. For effiency purposes permissions that equal zero are left undefined to keep the size of the model as small as possible.

4.1 Delta Datastructures

To keep track of changes in permissions a `Delta` datastructure is used. A `Delta` object keeps track of the changes in permissions that happen within a thread. Every `Delta` keeps track of only a single heap object or a single element of an array. These characteristics allow the basic `Delta` object to be as small and simple to calculate as possible. It is possible to aggregate multiple deltas together to create an overview of a heap object, an array or even an entire thread. An overview of the `Delta` and its aggregators can be found in figure 4. For generating delta objects and their aggregators a `DeltaFactory` has been implemented, this makes it possible to generate all deltas from a central point in the visualization tool.

In order to visualize the flow of the permission over the exection of the program the visualization tool calculates delta's using two `PermissionModel` instances: the model that represents the current programstate and the model that represents the programstate of the previous position in the execution. This previous position can actually be multiple interpreter steps ago, this allows for more flexibility in implementing and extending the interpreter as having multiple changes in a single step does not break the `DeltaFactory`. Delta calculations are done on demand.

4.2 Challenges

During the implementation of the prototype we stumbled upon two problems that turned out to be major complications.

4.2.1 Rational Numbers

Because permissions in VerCors are defined as rational numbers or fractions we have implemented a `Rational` datatype. This datatype stores the integer value of the numerator and the denominator. The difficult part of implementing the `Rational` is that it also had to support 0^+ and 1^-, as those values are also used for permissions. The complication occured when a bug caused the 0^+ value to be equal to the 0 value. This caused big problems since a permission of 0 means the thread has no access at all while a permission of 0^+ means a thread has read access.

4.2.2 Aliases

Threads can have several different objects which all reference the same object on the heap. Those different objects that reference the same heap object are called aliases. The existence of aliases complicates the generation of the permission model as you have to link multiple fields to the same permisssion. Initial implementations of the permission model did not take the existence of aliases into account. Therefore aliases had to be added later on which caused a very inconvenient loss of time.

4.3 Limitations

Because of the limited amount of time available for this research the choice was made to limit the initial prototype in scope. The prototype supports only simple heap objects: Integers, Booleans, Chars and arrays of those types. More complex structures such as objects and lists are not supported.

At the same time the prototype does not support the entire PVL language and only works for simple fork-foin and parallel structures.

5. VISUALIZATION

As said before, the permission flow of a program consists of permissions flowing from one thread to another. Such a system of permissions flowing from thread to thread can be modeled in the form of a directed graph where the nodes are threads and the edges are the permissions that change ownership. Using the delta factory it is possible to generate this graph for the entire program.

Using the Prefuse visualization library for Java [1] and its included graph datastructures it was possible to visualize some initial graphs. As can be seen in Figure 5 the vanilla library was not adequate for our purpose. Thread names are not rendered for the nodes, nor are the costs of the edges. At the same time multiple edges between two nodes, as is the case when threads exchange permissions on multiple fields are rendered on top of each other. For this reason we had to extend the Prefuse library with more advanced rendering methods, such as custom node and edge renderes which are capable of printing the required information and labels.

5.1 Overview

In accordance with the visualization mantra described by Shneiderman [7] we start by creating a proper overview of the permission flow. Such a overview should not contain too many details as it is only there to allow the user to identify nodes (which represent threads) of interest. To allow the user to do this, the user will have to know which threads have changes in permissions happening to them

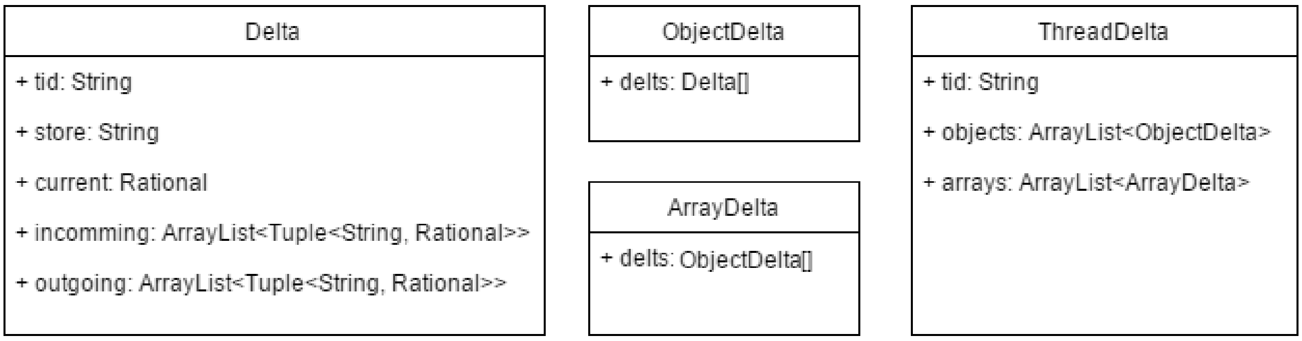
Delta
+ tid: String
+ store: String
+ current: Rational
+ incomming: ArrayList<Tuple<String, Rational>>
+ outgoing: ArrayList<Tuple<String, Rational>>
ObjectDelta
+ delts: Delta[]
ArrayDelta
+ delts: ObjectDelta[]
ThreadDelta
+ tid: String
+ objects: ArrayList<ObjectDelta>
+ arrays: ArrayList<ArrayDelta>

Figure 4. The Delta class and its aggregators.

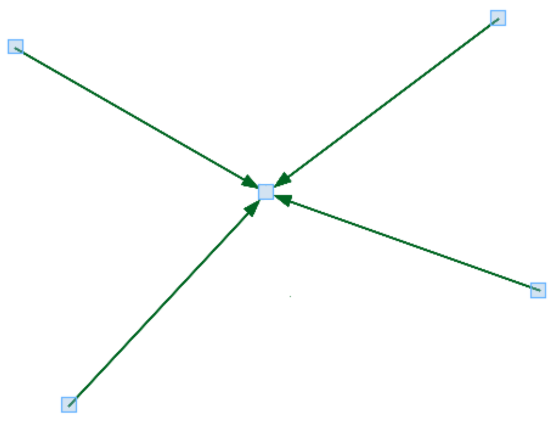

Figure 5. The permission graph with the default Prefuse renderers.

and which do not have that happen. It is also very important that the user gets an overview of the entire collection, this means the overview has to contain all threads, including those which have already finished running or those which are not yet running (but are initialised). Every node has a label with the thread identifier, this makes it easier for the user to identify special threads such as the "main" thread.

Because this overview contains a node for every thread that exists within the program we have decided to limit the amount of edges between two particular nodes to two; one edge in either direction. This greatly limits the amount of clutter in programs with a higher amount of threads. However with just the edges between the nodes the user does not have enough information to identify nodes of interest, because the user is not able to tell which permissions were exchanged. To solve this problem labels are added to the edges, those labels contain a list of all heap objects on which permissions flow in the direction of the edge. The names used in the list are dependent on the alias the giving thread knows them by. If the thread knows the heap object by multiple aliases all names are added to the list. An example of the overview from the prototype can be seen in Figure 6, which is the same graph as Figure 5 but now with custom rendering. In the overview field labels are not rendered to prevent clutter and to keep to overview as simple and clear as possible.

5.2 Zoom

Double clicking a thread in the overview zooms in on that particular thread. In this "Thread Display" the selected thread is used as a point of view. An example of such a display can be found in Figure 7. Since we use the central node as a viewpoint only the flow of permissions relevant to this node is visible. Every edge is connected to the correct object on the central node. Every edge has a label that corresponds to the value of the permission that flows from or to the central node. The current value of all permissions of the central node is rendered on top of the central node. Double clicking on a different node switches the viewpoint to that node. The visual variable color has been used to differentiate between incoming and outgoing permissions. At the same time the visual variable of size is used to differentiate between the amount that is transfered by different edges.

There is some future work left in the zoom functionality. The initial goal was to extend this zoom functionality to the heap objects. Double clicking on *var*0 in Figure 7 sopen a different display which should show the locations of all permissions on that heap object at that moment in time. The double click on the various heap objects is already detected by the event handling, all that is left to do is actually implement the display that should open.

5.3 Details On Demand

Even though assertion-based debugging is easier and more efficient than looking at source debugging information [6], some users of the tool might appreciate quick access to the actual values of various locals on the thread stack or

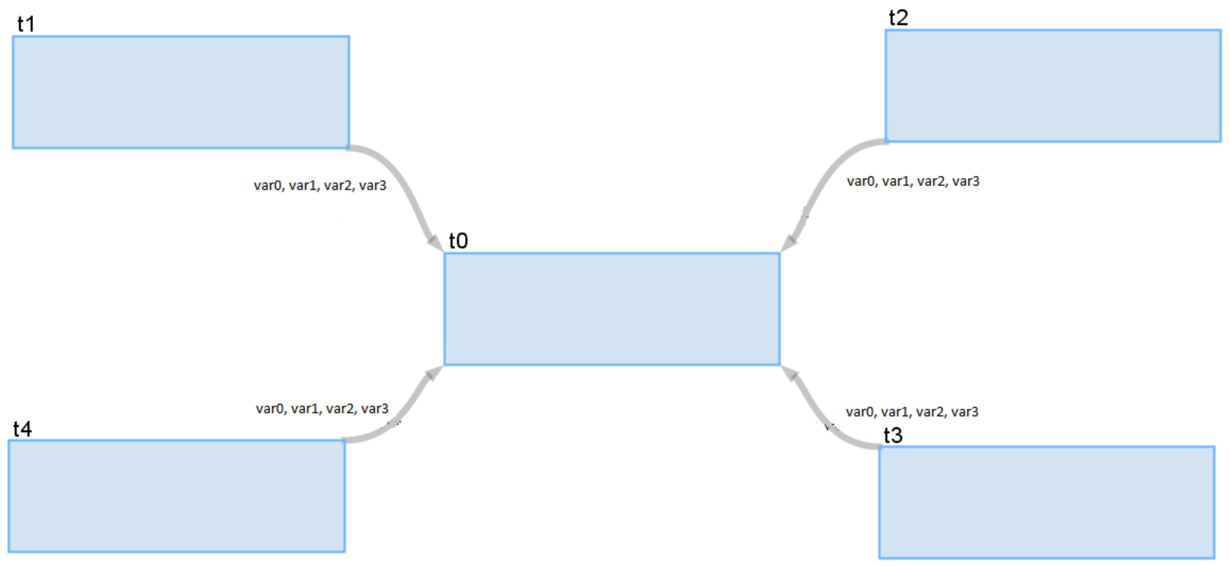

Figure 6. An example program overview.

objects on the heap. We looked into two different ways of giving the user access to these details.

The first possibility is showing the actual value of various fields on mouseover. The advantages of this method are its quick accessibilty and intuitiveness. The disadvantages are unwanted information and popups when the user moves the cursor.

The second possibility is allowing the user to open a table, either in a different view, or by adding it to the existing visualization, which contains the requested information. The biggest advantage of this method is the fact that it is truly on demand. The biggest disadvantage is that it either takes up some sizeable screen real estate when visualized next to the visualization in a different window or the more complex and time consuming implementation in the case of adding a table to the visualization as this table would need a different custom renderer.

A test conducted amongst a small amount computer science students resulted in a small preference for the tooltip solution, see table 1. Further investigation might prove useful before a solution is implemented.

Table 1. Details on demand results

Tooltip	**Table**
8	6

6. PROTOTYPE AND VALIDATION

A prototype visualization tool has been implemented as a result of this research. The prototype has been tested and validated using a paralel implementation of the Fibonacci algorithm (see Figure 8).

An example of the prototype GUI can be found in Figure 10. On the right a simple navigation panel can be found, the user can swap between views by changing their selection in this panel. The main visualization takes up the majority of the screen, this window can be resized to fit the user's preference. The green field has been reserved for a future code highlighting add-on. For more information about the prototype see Appendix A which walks through the execution of the fibonacci example.

7. CONCLUSION

The main research question of this paper was how to create a visualization method for the permission flow in concurrent programs. This goal has been achieved with the prototype visualization tool for the VerCors verification tool. To create this we first created an interpreter to gain access

```
// begin(all)
class Fib {
    static void main(){
        Fib f;
            f=new Fib(4);
            f.run();
        int res=f.output;
    }

    int input, output;

    requires Perm(input,1/10) ** Perm(output,1);
    ensures  Perm(input,1/10) ** Perm(output,1);
    void run() {
            if (input<2) {
                    output = 1;
            } else {
                    Fib f1;
                    f1 = new Fib(input-1);
                    Fib f2;
                    f2 = new Fib(input-2);
                    fork f1; fork f2;
                    join f1; join f2;
                    output = f1.output + f2.output;
             }
    }
    ensures Perm(input,1) ** //linebreak
             Perm(output,1) ** input==n;
    Fib(int n){
             input = n;
    }
}
// end(all)
```

Figure 8. Paralel implementation of the Fibonacci algorithm in PVL.

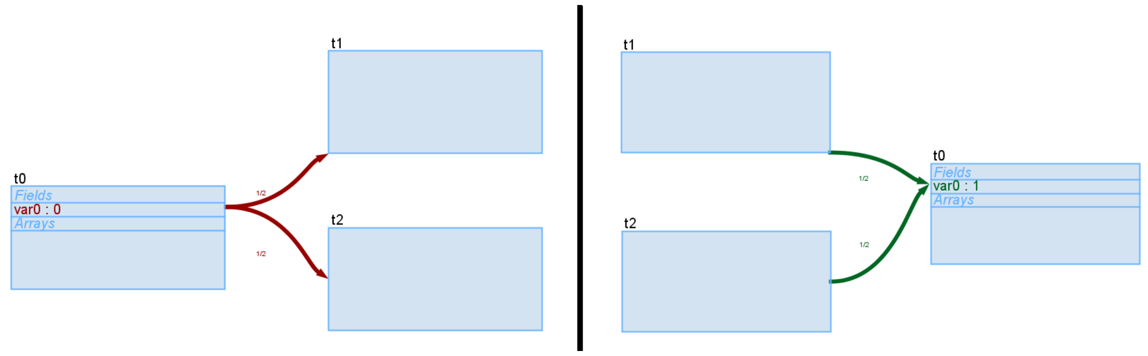

Figure 7. Thread t0 giving its permission on var0 to t1 and t2 (left) and receiving it back (right).

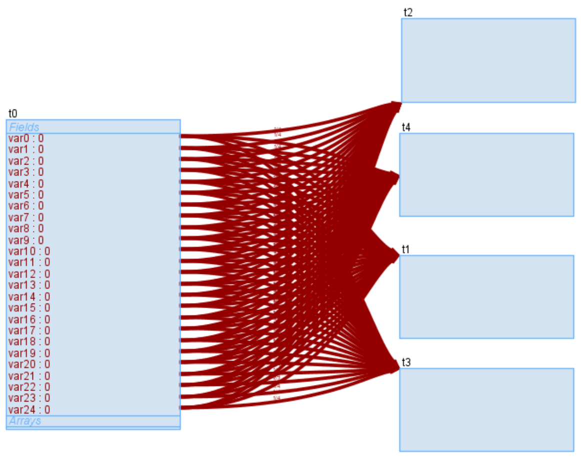

Figure 9. Many changes in a single step

to the required data. Using this data we were able to visualize the permissions of the threads within the program. Using delta calculations on the information retrieved form the interpreter we are able to calculate the changes of those permissions when compared with a previous point in the execution. We used the calculated delta's to visualize the changes in permissions as the edges of a directed graph of which the nodes represent the threads.

Because the prototype is very limited in its current state there is a lot of room for future work. The tool supports larger and/or more complex programs as long as the amounts of nodes and edges that have to be drawn are somewhat limited. Large amounts of data cause the visualization to stutter and, even worse, cause the visualization to become extremely unclear as edges and labels are overlapping eachother. This makes the entire graph almost unreadable as can be seen in figure 9. Luckily it is possible to control the amount of edges that are shown every step by making sure that every step is only a single interpreter step. This solution is not perfect, if an user wants to jump to a breakpoint or wants to make bigger steps in general it is no longer possible to assure that we only make a single interpreter step per visualization step.

7.1 Future Work

7.1.1 General Improvements

The prototype is a far from complete visualization tool. Not only does it support a limited subset of the PVL language it's also missing a lot of "ease of use" functionality such as, a resizeable window, a tidier layout, better transitions between different display mode and the camera not resetting to the start position when pressing the next step button.

7.1.2 Breakpoints

Since the visualization was meant to help programmers see the permission flow of their program, understand it and if necessary debug it, the addition of breakpoints could be very valuable. In order to achieve working breakpoints we would have to add breakpoint recognition to the interpreter. Once the interpreter recognizes breakpoints it should be possible to implement a "step till breakpoint" method. Bigger programs with either many threads or a lot of heap object will pose additional challenge.

7.1.3 More Complex Programs, Data And Structures

The current tool is compatible with integers, booleans, chars and arrays over those types. Extending this to include more complex structures such as lists and trees will eventually be necessary.

7.1.4 Code Highlighter

A code highlighter would highlight the lines of code in the source code that are currently active. This kind of

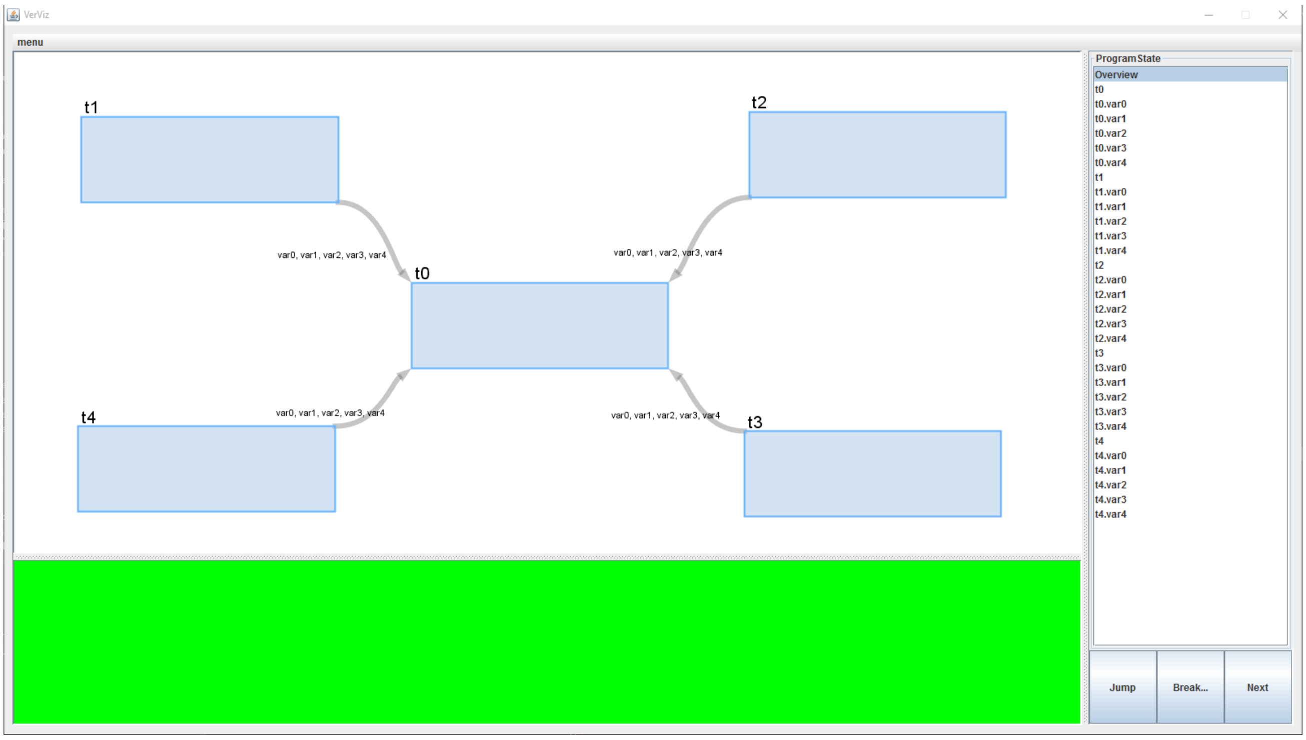

Figure 10. The prototype GUI

functionality is very common in source code debuggers. A code highlighter would be a valuable addition if we were to turn the visualization tool into a proper debugging tool.

8. REFERENCES

[1] The Prefuse visualization toolkit, 2012. http://prefuse.org/.

[2] A. Amighi, S. Blom, S. Darabi, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Verification of concurrent systems with vercors. In M. Bernardo, F. Damiani, R. Hähnle, E. B. Johnsen, and I. Schaefer, editors, *Formal Methods for Executable Software Models: 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, volume 8483, pages 172–216. Springer, 2014.

[3] M. Carpendale. Considering visual variables as a basis for information visualisation. department of computer science, the University of Calgary, 2008.

[4] S. Gijsen. Runtime permission checking in concurrent Java programs, 2015. MSC thesis University of Twente.

[5] K. R. M. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. Springer, 2009.

[6] L. G. Murillo, R. L. Bücs, D. Hincapie, R. Leupers, and G. Ascheid. Swat: Assertion-based debugging of concurrency issues at system level. In *The 20th Asia and South Pacific Design Automation Conference*, pages 600–605, Jan 2015.

[7] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*, pages 336–343, Sep 1996.

APPENDIX

A. FIBONACCI EXAMPLE

This Appendix shows snapshots of the prototype during execution of the fibonacci code in Figure 8. Only the forks and joins on the main (first) thread are shown.

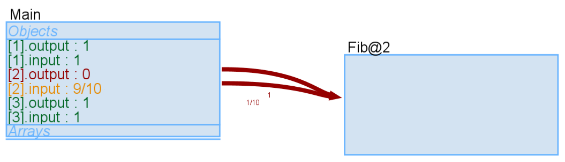

Figure 11. The main thread forking the first recursive step ($input - 1$).

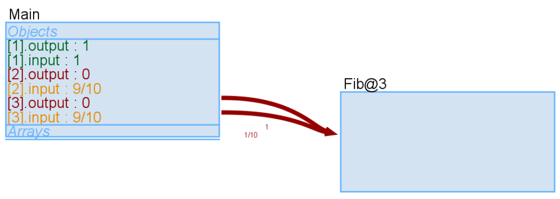

Figure 12. The main thread forking the second recursive step ($input - 2$).

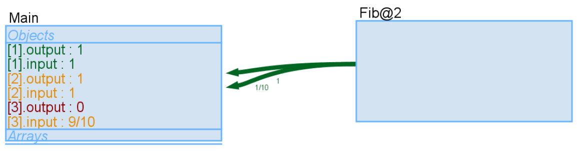

Figure 13. Main thread joining the first recursive thread. Regaining read access to its output, which has now been calculated.

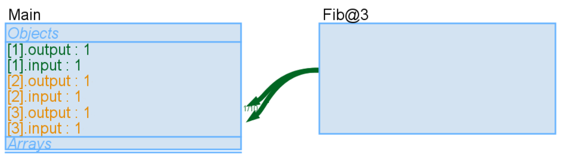

Figure 14. Main thread joining the second recursive thread. Regaining read access to its output, which has now been calculated. Both $input - 1$ and $input - 2$ are accessible now and the final answer can be calculated.