Static Visualization of Dynamic Data Flow Visual Program Execution

Buntarou Shizuki
Institute of Information Sciences and Electronics
University of Tsukuba
shizuki@is.tsukuba.ac.jp

Etsuya Shibayama
Graduate School of Science and Engineering
Tokyo Institute of Technology
etsuya@is.titech.ac.jp

Masashi Toyoda
Institute of Industrial Science
University of Tokyo
mtoyoda@acm.org

Abstract

We propose 'Trace View', a static visualization method for monitoring and debugging the dynamic behavior of programs written in data flow visual programming languages. Trace View presents a hierarchical structure of the data flow between nodes that is created over the execution time of the program. The view also serves as an interface that allows the programmer to select a data stream link when data must be examined during debugging. Moreover, since visualization grows in size according to the life time of the program, we have developed techniques to scale the view using a multi-focus focus+context view.

(a) (b)

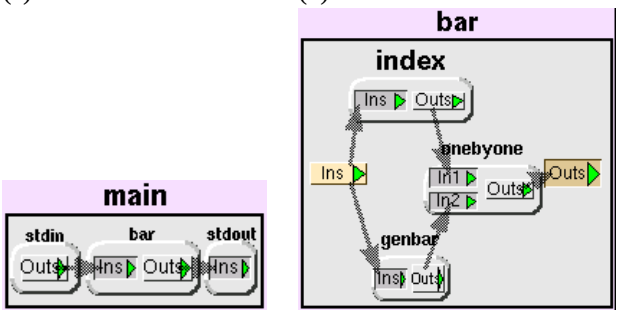

Figure 1. Sample data flow diagrams of declarative data flow VPLs

1. Introduction

Various animations that visualize program execution, such as algorithm animation (e.g., [12], [2], and [13]) and visualization of visual program execution (e.g., [8, 7] and [3]), have been proposed. The animations are generated by (1) defining mapping rules that map each state of a program into a key frame during its execution and by (2) interpolating the key frames. The result is an animation that represents the transition of states in the program as the execution proceeds.

Alghought such animation successfully provides a comprehensive overview of the execution and transition of the states of the target program, it can only present a trigger or its result of behavior of a component of the program since a frame of the animation is merely a snapshot of states of the program. Therefore, it is not sufficient as a debugging tool, since examination of causality between a trigger (input) and its result (output) is fundamental to validating the component.

Therefore, a visualization system must enable the programmer to examine both input to and output from a component simultaneously. Moreover, it is also required to provide functionality that enables the localization of possible bugs from global (abstract) to local (concrete).

In this paper, we proposes 'Trace View', a static visualization that fulfills the above requirements for monitoring and debugging dynamic behaviors of a program in data flow visual programming languages (VPLs). We also present a debugging methodology that uses the visualization.

2. Background

This section outlines the *declarative data flow VPLs*, their behavior, and bugs in these languages, as background to the discussion that follows.

2.1. Program

A program in a declarative data flow VPL is a collection of declarative rules that define *processes*. There are two types of processes: *composite processes*, which are visually defined in data flow diagrams, and *primitive processes*. A composite process is declared by a set of visual *composite rules*, each of which spawns child processes and forms a data flow network of processes with data flow links among

the processes. A guard may be attached to a composite rule to enable/disable the rule conditionally at runtime. A primitive process is defined in a certain way (e.g., as state transition diagrams or in textual languages).

Figure 1 shows examples of visual composite rules. Figure 1(a) is the rule of main, which is the data flow diagram consisting of three processes: stdin, bar, and stdout. Figure 1(b) shows the rule that defines bar, which consists of index, genbar, and onebyone. stdin and stdout are pre-defined processes that read and write the data stream at runtime, respectively; index reads data and outputs the corresponding indices for each input datum; genbar reads integers and generates strings, each of which consists of "x" characters whose length is the integer; and onebyone reads data from its two input ports (In1 and In2) one by one and concatenates both pieces of data as strings. (The definitions of these three processes are omitted in this paper.) As a result, if the standard input of the program is:

```
10 2 7 5
```

then, the program will produce the following text to its standard output:

```
1 xxxxxxxxxx
2 xx
3 xxxxxxx
4 xxxxx
```

2.2. Behavior

Execution of a program in a declarative data flow VPL consists of a sequence of parallel applications of composite rules from the root process (a composite main process), and execution of primitive processes. The application of a composite rule creates a *runtime data flow network*, which consists of *runtime processes* and *runtime stream links* between them as defined by the rule. A runtime process terminates when all of its immediate subprocesses have terminated. As a result of the dynamic creation and termination of processes during execution, the process network dynamically changes its topology at runtime as the execution proceeds. Since a runtime process (both composite and primitive) does not share any variables and memories with other processes, only the input determines the behavior of a runtime process (either composite or primitive).

The sample program in Figure 1 produces the process tree illustrated in Figure 2.2 at runtime.

2.3. Bugs

In this language, the programmer may observe the following four categories of abnormal execution due to bugs in the program: *incorrect success*, *abnormal termination*, *perpetual suspension*, and *infinite execution*. Incorrect success occurs when the execution succeeds, while the program produces output that the programmer considers incorrect. Abnormal termination occurs when faults (e.g., division by zero) occur in one or more runtime processes, thereby terminating the execution of the program. Perpetual suspension is the same as deadlock. Infinite execution occurs when one or more processes continue execution infinitely, probably due to an infinite loop or infinite mutual recursion.

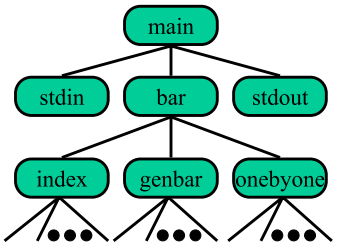

Figure 2. Process tree for the same program

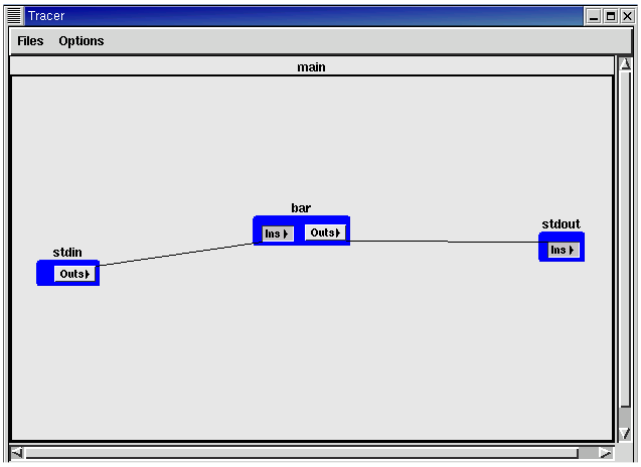

Figure 3. A snapshot of Trace View

2.4. Requirements for debugging

To detect and localize bugs, the following information must be provided.

Input and output of processes are vital to localization of bug(s). Since the bahavior of a runtime process is affected only by its input, the behavior can be validated by comparing its input and output.

Dependencies between processes are also crucial for localization. Data dependencies (i.e., runtime data stream links) lead the programmer near to location(s) of bug(s), when she/he finds improper data in them. The parent-child dependencies between processes are also fundamen-

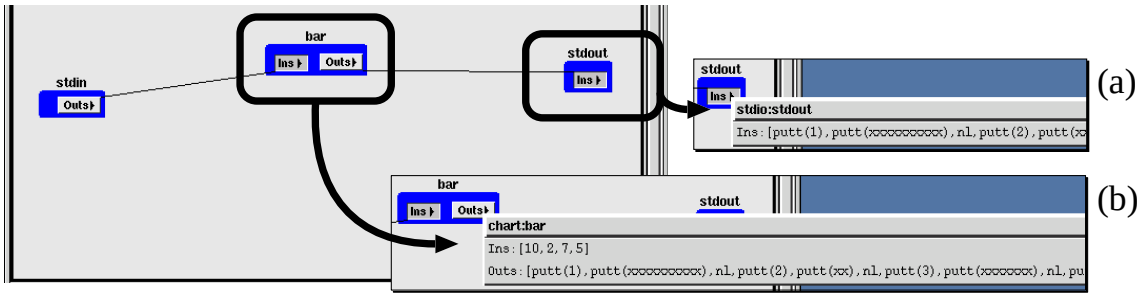

Figure 4. Browsing data for runtime processes

tal for narrowing down the possible location(s) of bug(s) from global (root) to local processes. If a process behaves wrong while all of its children behaves properly, bug(s) will be in the implementation of the parent process. Moreover, the dependencies are also important for distinguishing many instances of the same process that may run at the same time during execution, since the program in the target VPL is concurrent.

The *topology of a runtime data flow network* is useful. Since the application of a composite rule results in an incremental expansion of the hierarchical runtime data flow network, the topology is a result of the execution. Therefore, it provides rich information for detecting bugs. For example, if the topology contains problems such as the ones below, there are one or more bugs in the program:

- There are unexpected processes in the hierarchy.
- There is an unexpected number of instances of a certain process.
- The number of nests in the network is unexpected.

3. Trace View

This section proposes techniques to present the required information described in the previous section in a visual manner, with an interface that provides functions to modify the visual presentation to satisfy the programmer's requirements. Henceforth, we call the visual presentation with an interface as *Trace View*.

Figure 3 is a snapshot of Trace View, which shows the runtime network generated by the sample program in Figure 1.

3.1. Accessing process input and output

Trace View shows the runtime processes created during program execution as rectangles. It also shows each runtime process's input and output ports on the corresponding rectangle.

In Figure 3, Trace View shows that three runtime processes (blue rectangles) were spawned during execution: stdin, bar, and stdout. When the programmer moves the pointer on stdout and bar in Figure 3, the balloons shown in Figure 4(a) and Figure 4(b) appear, respectively, like a balloon tip. The balloons show the beginnings of the traces of the data, input to and output from the process, together with port names. This enables the programmer to easily browse through them by moving the pointer to a desired process.

3.2. Showing dependencies

Trace View represents data dependencies between processes as links, and parent-child relationships as nested rectangles. The view can be adjusted to show interesting parts of the hierarchy in detail or to hide unnecessary parts of the hierarchy.

In Figure 3, stdin, bar, and stdout are the children of main and form a runtime data flow network at execution. The view can be adjusted on a process basis. In the above example, when bar in Figure 3 is manipulated to show it in detail, the view changes into Figure 5. As a result, the view shows the detailed internal network of bar, which consists of index, genbar, and onebyone. This mechanism allows the programmer to examine a hierarchy from the top level to lower levels.

To allow close examination of the data transmitted through a port, a monitor is provided. Figure 6 is a monitor showing the content transmitted through the Outs port of onebyone. The programmer can examine its entire contents by using the scroll bar. A button label as "check" is provided to check which process this port belongs to. When this button is pressed, the corresponding port starts to blink.

3.3. Showing topology

To provide the programmer with rich information about topology of the runtime data flow network, Trace View has

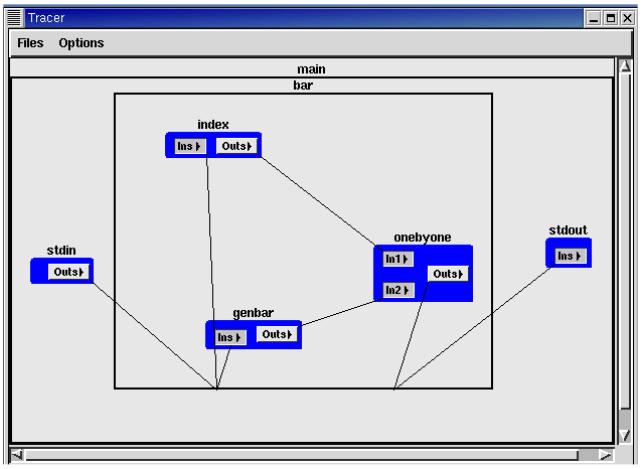

Figure 5. Showing deeper networks in the hierarchy

Figure 6. Examining data

a mode in which the view tries to show the topology as in detail as screen space permits. For example, the view tries to show the internal network of bar like Figure 5 whenever it is possible in the mode.

4. Localization methodology

The methodology for localization of bug(s) using Trace View is similar to the algorithmic debugging proposed by Shapiro [10]. Namely, the methodology mainly consists of the following three procedures.

- Validating a process (gray rectangle P in Figure 7(a)) by monitoring the history of its input ($In_i, i = 1...k...$) and output ($Out_j, j = 1...h...$).
- Localizing a bug by following data links back to a process outputting improper data, based on data dependency. For example, if we find that In_1 of P in Figure 7(a) contains erroneous data, we can examine P_1 (shown in Figure 7(b)) next.
- Localizing a bug by validating the children of a process if the process proves buggy. For example, if we find that all the data in In_i are correct for all i, but P produced erroneous data via Out_h for a given h in Figure 7(a), we can check the children ($P_h, h = 1...S_i...S_j...S_k...$) in Figure 7(c). Note that this check should begin by examining the children producing erroneous data directly, to enable us to approach the location of bugs quickly, without checking all the children. For example, P_{S_k} should be checked first when Out_h contains erroneous data, as shown in Figure 7(c). If all the children behave correctly, we can conclude that the implementation of the parent process contains bugs.

For each type of bug, debugging starts with the following processes: the processes that directly produced the incorrect result leading to incorrect success, processes that caused an abnormal termination, the suspended processes in perpetual suspension, and the processes that continue execution infinitely.

5. Implementation

We have implemented Trace View as the visual debugger in our visual programming environment KLIEG[16].

5.1. Constructing a static view

Since examining data transmitted through runtime data stream links is vital to the method, it is important that the programmer is able to quickly display and examine data.

To this end, we first store all the data transmitted through all the data stream links at runtime. Next, we construct a static view that shows entire hierarchies, which contain all the runtime processes and runtime stream links.

Note that the word “static” means that the view presents processes that have already terminated computation and all the data they transmitted in one view. This allows examination all the processes and data that were created from the beginning of execution, without re-execution. It is possible to show the same quantity of information using animation; however, in order to validate processes by comparing the input/output, this often requires information to be remembered, since each frame of the animation replaces previous frames and the information shown by the previous frames disappears from the screen. This leads to re-execution.

5.2. Applying a focus+context viewing algorithm

Since screen space is limited and the hierarchy of a runtime network becomes broad and deep as the execution proceeds, we must select processes to be displayed. A pan+zoom interface is a partial solution to this problem. This scales the original network linearly and provides scrollbars that allow the programmer to move to part of the scaled view. However, the programmer may lose track of the current location when zooming in on a portion of the

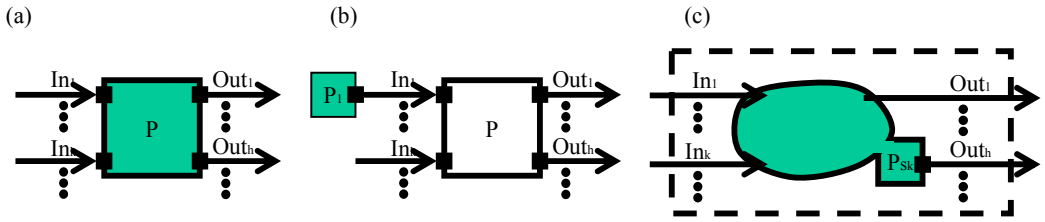

Figure 7. Localization method

runtime network, particularly when many subnetworks are instantiated from one rule and are consequently similar.

In contrast, the focus+context approaches that Furnas[6] pioneered are powerful for navigating such networks, and many variants have been developed. In particular, the Continuous Zoom[1] has the following properties, making it a suitable algorithm. First, the algorithm supports a multiple focal point facility. Second, it guarantees the presence of paths to every network node (i.e., process), even to hidden nodes. This allows the programmer to examine every node. Finally, the algorithm preserves the nested structure of the network and provides a navigation interface based on that structure. Consequently, the algorithm reveals caller-callee relationships between processes in the process network hierarchy, such as depicted in Figure 5. The Continuous Zoom algorithm can be applied simply by treating each process (i.e., the rectangles depicted in Figure 5) as a node of the algorithm.

5.3. Smooth transition of the view

So that the programmer is not confused by abrupt transitions of view, we animate the transitions smoothly.

6. Related work

One of the most popular data flow VPL system is Prograph [4]. In Prograph, the programmer can display the runtime data flow networks of a program graphically using multiple windows. In a window, a selected part of the networks can be shown. However, the entire structure of runtime data flow networks cannot be grasped directly; since only a limited part of the network is displayed in a window, the information displayed in separate windows must be integrated mentally. There are also visual debuggers that show runtime data flow networks as a user interface to examine the execution of non-visual programming languages, such as HyperDEBU [14].

Visualizing traces of program execution has been researched and some systems have been created[9][5][15]. However, these systems do not provide tools that localize bugs directly from the visualized traces.

7. Discussion and future works

By updating the view each time a composite rule is applied, it is easy to extend Trace View to show the control flow of the execution. This shows changes in the topology of the runtime data flow network. Moreover, in order to help the programmer discover bottlenecks within the network, coloring each runtime process according to show the scheduling status is a possible extension. We have already incorporated these extensions into our visual debugger[11] and find them quite useful for monitoring the behavior of programs, especially those in an early stage of development, since this allows the programmer to start debugging the instant that suspicious behavior is observed in the view.

In algorithmic debugging, the order of the number of questions that a programmer has to answer to localize a bug is $log(n)$, where n is the number of goals. In the VPL described in Section 2, the situation is similar, that is, the number of goals in algorithmic debugging corresponds to the number processes that are created during execution.

In contrast to the traditional approach of algorithmic debugging, in which the programmer must answer "Divide and Query" questions that the system imposes, our proposed method (described in Section 4) leaves all of the questioning to the programmer. That is, a programmer must consider questions such as "what should I check next to localize a bug?" and "Does this process behave properly?" This puts the programmer at risk of asking increasing numbers of questions if the programmer adopts the wrong strategies while examining the program. However, our debugging methodology in combination with our visualization which gives an overview of the execution has the potential to improve the order, because the rich information provided by visualization helps programmers find the approximate location of bug(s) and also helps them make a decision about the direction to make to localize the bug(s). Therefore, integrating an interface appropriate for conducting algorithmic debugging with our proposed Trace View interface retains the advantages of both method, and is a future avenue of this research.

Although we believe that the interface gives us great scalability in debugging, the current implementation stores

all the data transmitted through runtime data stream links. This limits the debugger's scalability. However, it is not necessary to preserve all the data. Preserving sufficient data to re-construct by a replay is a possible choice. To implement this, when a window is open to show details of the internal data flow network, the system re-executes with the input and reproduces all the data in the internal data flows. As a result, the programmer can examine the details of the internal data flow as though all the data were preserved by the system.

However, if re-execution requires substantial computations, the programmer must wait until it terminates. This interferes with the programmer's debugging tasks. A challenging research topic is to develop algorithms that store sufficient information at runtime to enable the system to recover data flows in a limited time, thereby not frustrating the programmer, while simultaneously minimizing the storage requirements.

8. Conclusion

This paper presents Trace View, a visualization technique that helps programmers to monitor and debug programs in declarative data flow VPLs. Trace View visualizes the history of the input/output of every process, including terminated processes, the hierarchical structure of processes, and data flow between processes in one view. The visualization also encourages programmers to detect buggy behavior in the execution. We have implemented Trace View as the debugger in our visual programming environment KLIEG.

References

[1] L. Bartram, A. Ho, J. Dill, and F. Henigman. The Continuous Zoom: A Constrained Fisheye Technique for Viewing and Navigating Large Information Space. In *Proceedings of the ACM Symposium on User Interface Software and Technology*, pages 207–215. ACM Press, Nov. 1995.

[2] M. H. Brown. Zeus: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of 1991 IEEE Workshop on Visual Languages*, pages 4–9. IEEE Computer Society Press, Oct. 1991.

[3] W. Citrin, M. Doherty, and B. Zorn. The Design of a Completely Visual OOP Language. In A. G. Margaret Burnett and T. Lewis, editors, *Visual Object-Oriented Programming: Concepts and Environments*, chapter 4, pages 67–93. Manning Publications Co., 1995.

[4] P. T. Cox and T. J. Smedley. A Visual Language for the Design of Structured Graphical Objects. In *Proceedings of 1996 IEEE Symposium on Visual Languages*, pages 296–303. IEEE Computer Society Press, Sept. 1996.

[5] D. Fahrenholtz and V. Haarslev. Visualization of STRAND Processes. In *Proceedings of 1995 IEEE Symposium on Visual Languages*, pages 114–115. IEEE Computer Society Press, Sept. 1995.

[6] G. W. Furnas. Generalized Fisheye Views. In *Proceedings of ACM CHI'86 Conference on Human Factors in Computing Systems*, pages 16–23. ACM Press, Apr. 1986.

[7] K. M. Kahn. Concurrent Constraint Programs to Parse and Animate Pictures of Concurrent Constraint Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 943–950. ICOT, June 1992.

[8] K. M. Kahn and V. A. Saraswat. Complete Visualizations of Concurrent Programs and their Executions. In *Proceedings of 1990 IEEE Workshop on Visual Languages*, pages 7–15. IEEE Computer Society Press, Oct. 1990.

[9] H. Koike and M. Aida. A Bottom-Up Approach for Visualizing Program Behavior. In *Proceedings of 1995 IEEE Symposium on Visual Languages*, pages 91–98. IEEE Computer Society Press, Sept. 1995.

[10] E. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[11] B. Shizuki, M. Toyoda, E. Shibayama, and S. Takahashi. Visual Patterns + Multi-Focus Fisheye View: An Automatic Scalable Visualization Technique of Data-Flow Visual Program Execution. In *Proceedings of 1998 IEEE Symposium on Visual Languages*, pages 270–277, Sept. 1998.

[12] J. T. Stasko. Tango: A Framework and System for Algorithm Animation. *IEEE Computer*, 23(9):27–39, Sept. 1990.

[13] S. Takahashi, K. Miyashita, S. Matsuoka, and A. Yonezawa. A framework for constructing animations via declarative mapping rules. In *Proceedings of 1994 IEEE Symposium on Visual Languages*, pages 314–322. IEEE Computer Society Press, Oct. 1994.

[14] J. Tatemura, H. Koike, and H. Tanaka. A performance debugger for parallel logic programming language fleng. In *Theory and Practice of Parallel Programming: Proceedings of the International Workshop TPPP'94*, volume 709 of *Lecture Notes in Computer Science*, pages 284–299. Springer-Verlag, Nov. 1995.

[15] E. Tick. Visualizing parallel logic programs with VISTA. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, pages 934–942, June 1992.

[16] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, and E. Shibayama. Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment. In *Proceedings of 1997 IEEE Symposium on Visual Languages*, pages 76–83, Sept. 1997.