The KISS Principle applied to Dataflow Language Paradigms for Visualization Frameworks

Hans-Peter Bischof, Swathi Annamalai

Rochester Institute of Technology
Center for Computational Relativity and Gravitation
102 Lomb Memorial Dr., Rochester, NY 14623
{hpb,sxa9002}@cs.rit.edu

Abstract. Most visualization frameworks and visualization systems are using dataflow languages as a programming paradigm. This paper will contrast the different paradigms and propose a successfully implemented and used KISS (*Keep It Simple Stupid*) approach for visualization frameworks. A dataflow language needs a runtime environment for execution. We will present a runtime environment based on a minimalistic dataflow language design. An example will illustrate our approach with a few selected examples.

Key words: Dataflow Languages, Visualization Frameworks

1 Introduction

This paper describes a successful adaption of the KISS principle to the design and development of a visualization framework. In this paper we will first discuss the use cases for dataflow languages and classify them. Then we will distill the minimal requirements of dataflow languages for visualization frameworks. Finally we present our design of a minimalistic dataflow environment, followed by an example.

2 Dataflow Languages

Dataflow languages are not new. A short history of dataflow languages can be found in Whiting's et all[11] paper from 1994. More recent history shows that dataflow languages are generally not used for generic programming purposes. They are only used, and useful, if data becomes the main concept behind a program. They provide a different way of how we look at computing. Von Neumann inspired imperative language styles think in terms of control flow on a line by line basis. In dataflow languages the primary focus is on the data moving through the system.

The building blocks of dataflow languages are nodes which are connected via a directed graph. The data is flowing between the nodes along the directed graph during the execution of a program.

One of the original design goals of dataflow languages was to make parallel programming and execution easier[9]. This goal is achieved by the property that nodes do normally not share any state information with other nodes. This property of dataflow languages naturally allows to parallelize or distribute dataflow programs.

A dataflow program is a directed graph. The nodes are computing components, and the edges are the dataflow paths. A node in a dataflow program has well defined input and output communication endpoints which are used to connect the nodes with each other. The communication endpoints may be typed in order to allow only matching connections, which reduces the amount of possible programming errors. Nodes should be able to process arguments which allow to fine tune the functionality of a them.

A dataflow language is typically executed in an execution engine. The execution engine controls the execution of the components based on the dataflow between the nodes.

The processing of data can be done in streaming or blocking mode. Kahn[5] proposed 1974 that the reading on an empty input communication endpoint will block the node from execution, until data becomes available. As a consequence, processing of the data by a node can only start when all input communication endpoints are valid, meaning presented with data.

Some dataflow languages can be programmed via a graphical programming environment. Dataflow programs then are simply written by creating nodes, connecting input and output communication channels, and assigning argument values. Figure 1 shows one example of a graphical representation of a dataflow program. The node *Stars* reads the file named *sim.dat* specified as an argument and sends the data to the node *Stars3D*, and from there the data is sent to the last node in the graph, a *Camera*.

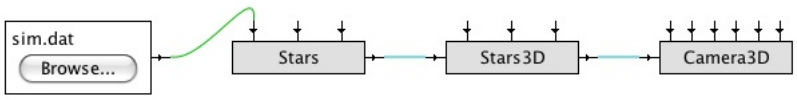

Fig. 1. Hello World

The Unix[8] shell[7] *sh* has been one of the first environments which successfully adopted a dataflow approach. The following UNIX pipeline sequence can be seen as a very simple example of a dataflow program.

```
cat file | sort -nr | head -5
```

The shell, the executing engine, while executing this pipeline sequence, has to create and start the processes, create the communication buffers, and connect standard in with standard out of the different processes. The shell knows nothing about the functionality of the nodes, or the type of data exchanged, but it is responsible for the dataflow from node to node and for the scheduling of the execution of the individual nodes.

A distributed version of this command sequence would look like the following:

```
cat file | ssh biggie sort -nr | head -5
```

The *sort* command will be executed on the host named *biggie*; the other commands will be executed locally. What has to happen in order to execute this program sequence is rather difficult from a shell point of view. Network communications need to be made, buffers need to be created, and two OS schedulers have to work in unison etc. Nevertheless the use case is rather simple and the developer of *sort* designed and implemented the command without imagining a use case like this. The ease of use and the simplicity of adding new components comes from the fact, that the operating system is responsible for the communication aspects.

The rest of this paper discusses a short history of dataflow languages and describe a minimalistic dataflow language for visualizations frameworks.

3 Discussion of Dataflow Languages

General purpose programming languages are not well suited to design solutions for dataflow based problems. A dataflow program is a directed graph, where the node act on the data they receive, and the edges connect the nodes. A node may have more than one incoming edge, and may have more than one outgoing edge.

Concurrencies of processes are implicit and controlled by the system in dataflow models[1]. Therefore they are less prone to errors due to deadlocks within the system components. Originally developed to exploit parallelism, dataflow languages are created with a need to appropriately combine data, task and pipeline parallelism to accomplish maximum performance. Granularity as a measure of dividing workload amongst system components is applied to dataflow models as fine grained and coarse grained dataflow.

Dataflow languages begun to appear in the late 1970s with TDFL (Textual Dataflow Language) being the first of the kinds to be developed using the graph based programming concept. LAU[4], VAL[6], SISAL[3] were some of the other dataflow languages that emerged using dataflow properties such as single assignment, conditional evaluation, iterative constructs, etc. LAB VIEW[10], a popularly used program development application has built-in scheduler that exploits parallelism due to multi-threading and multi-processing properties of the language. Commonly used in Digital Signal Processing applications for chip design, LAB VIEW components are structured as wires depicting propagation of variables and operations (nodes) are specified by sets of instructions.

We will discuss the requirements for dataflow languages in the next chapter.

4 Discussion of Requirements for Dataflow Languages for Visualization Frameworks

As a first requirement for dataflow language, it must be possible to create the dataflow graph, i.e., this means it must be possible to create the components and connect the data communication endpoints with each other. Additionally it must be possible to fine tune the behavior of the components during execution in order to make a dataflow language program useful.

From here on, two possible paths can be chosen. The first one is a highly sophisticated dataflow language, which supports the use of variables, control structures, function calls, etc. These languages allow to write complete data flow programs.

As an example, Iris Explorer [2] is using *skm* as the scripting language. Skm is a general purpose language and has roughly 50 commands and is not too hard to learn but very difficult to master. A note from IRIS Explorer User's Guide (007-1371-030.1994): *"This chapter describes how to use the scripting language, Skm, to take full advantage of the command interface in Explorer."* This means, in order to use, or create additions to Iris Explorer you have to fully understand skm. Figure 2 shows a simple code snippet from the handbook (www.rc.yale.edu/userguides/graphics/explorer/html/doc/ug/ch6.htm):

```
(define image-conn
 (procedure ( mod1 mod2 )
 (connect mod1 "Img Out" mod2 "Img In")))
```

Fig. 2. Simple example of the Skm scripting language

Within this approach languages like skm must be learnt in order write visualization programs, which means each developer goes through a steep learning curve. The property that nodes do not change state information of other nodes has to be kept in order to make it possible to distribute the program over a network. A language must not leave this up to the developer, the language must enforce this property. The execution engine, which has to support all provided features is rather complex, and therefore difficult to develop and to substitute. Adding new components requires the complete knowledge and side effects of the dataflow language.

The question to ask here is, what makes an environment useful and accessible to a large group of users/developers.

The other choice would be a simple language, which supports the creating of components, connecting them to a directed graph, and supplying the components with arguments. The control structures are implemented in components using commonly known languages, like Java, etc.

The major difference between the two choices is where to put the complexity of the program. The first choice allows to put the complexity in the language; the

second choice will force to put the complexity in the building block, the node. The second choice will enforce the creating of reusable, simple, building blocks used in a straight forward dataflow environment.

The rest of the paper will describe a visualization framework named *Spiegel* [1]. A minimalistic approach was chosen as a design principle for this framework. Spiegel is developed in Java and uses a dataflow language name *Sprache*[2]. This language has 8 keywords and can be learned in 5 minutes.

5 Minimalistic Design of Dataflow Environments for Visualization Frameworks

Our solution for the dataflow environment consists of 3 components: a simple language *Sprache* to define the dataflow graph; a programming interface which each component has to implement; and a runtime engine which executes the dataflow program.

Sprache has 8 keywords, which allows to create components, connect them, set default values and create functions. The following code snippet shows an example, leaving out the CLASSPATH for easier readability:

```
function new     extractor  Stars                    # defining of a component
function set     Stars      blackHoleSize 3          # setting a value
function connect BH_extractor output stars to BH_visualizer input stars
                                                     # connection of
                                                     # communication channels
```

Fig. 3. Sprache: Dataflow Language Example

A component, developed in Java, which can be used in the Spiegel framework has:

- k named and typed data input communication endpoints, $0 \leq k$
- l named and typed data output communication endpoints, $0 \leq l$
- m named and typed argument inputs
- 1 command channel endpoint
- to implement a programming interface, which requires to implement one method *update()*

The method *update()*, will be called if one of the input communication endpoints or an argument has an updated value. The execution engine is responsible for invoking the method, providing the values from the input communication

[1] Spiegel is the German word for mirror. Like the mirror in a telescope helps to observe the universe, Spiegel helps to observe the simulation of objects in space.

[2] Sprache is the German word for language. Both names were chosen by Jonathan Coles.

endpoints, passing along the arguments, and sending the data along the communication paths.

6 A Camera Follows the Path of a Black Hole During a Merger Process

As a demonstration of its ease of use We would like to use the Spiegel system to solve the following problem: A data set contains the paths of three black holes during a black hole merger process. For the visualization, a camera should be fixed at a given position in space, but follow the motion of a specific black hole. The problem here is, that the dataflow needs to be analyzed during the visualization process. Figure 4 shows a graphical representation of the Sprache program; Figure 5 shoes the code of the program. The *BH_extractor* reads the position of the black holes and passes this information to the *BH_visualizer* and *PositionOfBh*. *PositionOfBh* extracts the position information of black hole number 3, and sends the position information to the *BH_Cam*. The position of the black hole will be updated with each time step, therefore the camera follows the position of the black hole.

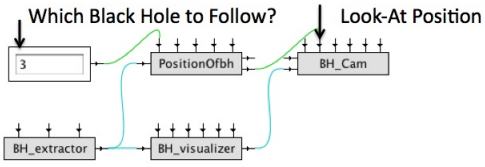

Fig. 4. Follows the Path of a Black Hole During a Merger Process

The first five lines create the 5 components, the value 3 is assigned to a component in line six, and the last four lines connect the components.

This program contains all 8 keywords and shows that even though the language is simple, interesting visualization systems can easily be described.

A component will be automatically executed on a remote machine, if a host name is supplied. The program shown in Figure 6 the component *BH_visualizer* is supplied with the host name *biggie*. The execution engine will in a transparent way to the component transfer the classes needed to *biggie*, create the needed communication channels, and will perform the execution. Figure 6 is the dataflow equivalent of

```
cat file | ssh biggie sort -nr | head -5
```

```
function new UserFunction BH_Cam                                          # 1
function new Stars BH_extractor                                           # 2
function new PositionOfThisOne PositionOfbh                               # 3
function new ConstantInteger ConstantInteger                              # 4
function new Stars3Dvolume BH_visualizer                                  # 5

function ConstantInteger set value 3                                      # 6

function connect BH_extractor output stars to PositionOfbh input stars    # 7
function connect BH_extractor output stars to BH_visualizer input stars   # 8
function connect PositionOfbh output position_2 to BH_Cam input location  # 9
function connect ConstantInteger output value to PositionOfbh input thisOne  # 10
```

Fig. 5. Sprache: Dataflow Language Example for Following the Path of a Black Hole

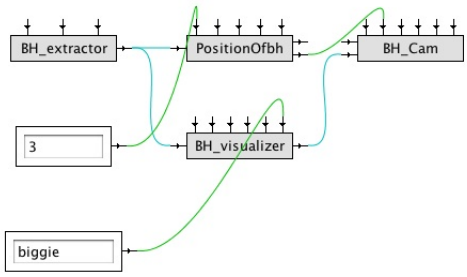

Fig. 6. Follows the Path of a Black Hole During a Merger Process

7 Conclusion

We presented a minimalistic dataflow language and runtime engine, which are easy to use, easy to understand, and easy to extend. Everything, which does not concern the dataflow, in other words all control flow decisions are done in Java. This allows a large group of developer to use the system with a two hour learning curve.

We lately successfully added an audio component to the system, which allows us to create a stereo audio representation of the data. The authors know of no other visualization framework where this is doable with little effort. The presented framework is a system where the KISS principle was successfully adopted.

Acknowledgements

We would like to express our thanks to Manuela Campanelli, Carlos Lousto and Yosef Zlowocher, all RIT, for their help and fruitful discussions. This team provided the data used and the interpretation of the visual representation.

References

1. J. Dennis. A Language Design for Structured Concurrency. The Design and Implementation of Programming Languages. *Lecture Notes in Computer Science*, 54:23–42, 1977.
2. David Foulser. Iris explorer: a framework for investigation. *SIGGRAPH Comput. Graph.*, 29(2):13–16, 1995.
3. J. Gaudiot, W. Bohm, T. DeBoni, J. Feo, and P. Miller. The sisal model of functional programming and its implementation, 1997.
4. John R. Gurd, John R. W. Glauert, and Chris C. Kirkham. Generation of dataflow graphical object code for the lapse programming language. In *CONPAR '81: Proceedings of the Conference on Analysing Problem Classes and Programming for Parallel Computing*, pages 155–168, London, UK, 1981. Springer-Verlag.
5. G. Kahn. The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475, 1974.
6. James R. McGraw. The val language: Description and analysis. *ACM Trans. Program. Lang. Syst.*, 4(1):44–82, 1982.
7. D. M. Ritchie and K. Thompson. The unix time-sharing system. *Communications of the ACM*, 17:365–375, 1974.
8. Dennis M. Ritchie and Ken Thompson. The unix time-sharing system. *Commun. ACM*, 17(7):365–375, 1974.
9. William Robert Sutherland. The on-line graphical specification of computer procedures, ph.d. dissertation. 1966.
10. Jeffrey Travis and Lisa K. Wells. *LabVIEW for Everyone with Cdrom.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
11. Paul G. Whiting and Robert S. V. Pascoe. A history of data-flow languages. *IEEE Ann. Hist. Comput.*, 16(4):38–59, 1994.