A Graph Editing Framework for the StreamIt Language

by

Juan C. Reyes

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Computer Science and Engineering

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2004

Author .....
Department of Electrical E Science
0, 2004

Certified by .....
singhe
ociate Professor
iesis Supervisor

Accepted by .....
rthur C. Smith
Chairman, Department Committee on Graduate Students

A Graph Editing Framework for the StreamIt Language

by

Juan C. Reyes

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2004, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Computer Science and Engineering

Abstract

A programming language is more useful if it provides a level of abstraction that makes programming more intuitive and also allows the development of tools that take advantage of the language's internal representation. StreamIt, a language for the development of streaming applications, has a hierarchical and structural nature that lends itself to a graphical programming tool.

I created a prototype StreamIt Graph Editor (SGE) to facilitate the development of streaming applications using StreamIt. The SGE provides intuitive visualization tools that allow developers to work more efficiently by automating certain processes. Thus, the programmer can focus more on design issues than on low level details that slow down the development process.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

I would like to thank my thesis advisor, Saman Amarasinghe, for giving me the opportunity to work on an exciting project. I have really enjoyed developing the StreamIt Graph Editor. Many people from the StreamIt group were instrumental in the coming together of this project and I would like to thank them all: Jasper Lin, Bill Thies, Kimberly Kuo, and Michal Karczmarek. In particular, I am grateful to Rodric Rabbah for his guidance in this project. He greatly contributed in determining the project's direction and in fixing various problems.

A special thanks to my academic advisor, Dennis Freeman. His good advice allowed me to make the right decisions during my MIT career.

My deepest gratitude goes to all my friends from MIT and Mexico who have always been there for me during the past five years. They have been essential in maintaining my sanity. I will always treasure all the memories from my MIT days.

I am grateful to all my relatives in Mexico who always believed in me. Most importantly, I want to thank my parents, Roberto and Aida, my sisters, Erika and Darlenne, my grandma, Carmen, and Tito. Your encouragement, love and support has been what has driven me to always do my best. I am lucky to have such a great family.

Contents

1 Introduction **11**

1.1 Overview 12

1.2 Organization 14

1.3 Motivating Example 14

2 Background **17**

2.1 StreamIt 17

2.1.1 Language Specifics 18

2.2 Related Work 21

3 A Graphical Editor for StreamIt **23**

3.1 Intuitive Visualization of Components 23

3.2 Alternate Perspectives 26

3.2.1 Editing Perspective 26

3.2.2 Overview Perspective 26

3.2.3 Hierarchy Perspective 26

3.3 Graphical Editing 27

3.4 Property Modification 28

3.5 Graph Layout 29

3.6 Display Tools 30

3.7 Editing Tools 31

3.8 Merging 33

3.9 Image Export 33

3.10 Code Creation 34

4 Implementation of the StreamIt Graphical Editor 35

4.1 General Overview 35

4.1.1 Eclipse 35

4.1.2 JGraph 36

4.1.3 SGE Internal Representation 36

4.2 Features 36

4.2.1 Creating StreamIt components 37

4.2.2 Connecting Components 38

4.2.3 Changing Properties 39

4.2.4 Expanding/Collapsing 40

4.2.5 Zoom 41

4.2.6 Visibility of Containers 42

4.2.7 Copying/Pasting/Deleting 43

4.2.8 Graph Layout 43

4.2.9 Merging 44

4.2.10 Specialized Splitjoin Functionality 45

4.2.11 Template Code Creation 45

4.2.12 Exporting Graph Images 47

4.3 Future Work 48

5 Concluding Remarks 49

A Code 51

B Figures 53

List of Figures

1-1 Merging splitjoin A and splitjoin B to produce splitjoin C 15

2-1 Container structures supported by StreamIt. 19
2-2 Hierarchical nature of components 20

3-1 Component representation in SGE. 24
3-2 Different perspectives of the stream graph 25
3-3 Hierarchy Perspective 27
3-4 Pipeline Layout 29
3-5 Splitjoin Layout 30
3-6 Feedback Loop Layout 30
3-7 Expanding/Collapsing Components 32

4-1 SGE Prototype 37
4-2 Toolbar 37
4-3 Property Window 39
4-4 Zoom (using hierarchy perspective) 41
4-5 Visibility of containers 42
4-6 Graph Layout 44
4-7 Sample Template Code. 46
4-8 Graph corresponding to template code sample 47

A-1 Source code for the EchoEffect StreamIt program. 52

B-1 Graph representation of the EchoEffect program 54

Chapter 1

Introduction

Software systems have become increasingly complex as the demand for better software has increased. As a result, software developers need to meet tight deadlines while at the same time construct these complex systems. In order to fight this problem, new tools to create software are required. These tools must be so simple and powerful that the software nearly writes itself.

The process of programming should be intuitive and automated. According to Charles Simonyi, the lead developer of the first WYSIWYG word-processing editor, "software should be as easy to edit as a PowerPoint presentation" [12]. The same way that PowerPoint slides are created by dragging components and setting properties, a graphical programming environment can be used to produce code faster and with less errors.

The ideal programming tool should have an intuitive graphical interface to simplify the software development process. James Gosling, inventor of Java, proposes a modeling tool that represents existing code graphically rather than with lines of code [12]. To produce an ideal programming tool, the language used to develop an application must resemble the design. Developers should be able to create high-level designs of what they want their programs to do. The developer should only have to modify the graphical representation of the application (something resembling a flow chart) instead of changing the actual code. Thus, the developer can focus more on design issues.

The ideal programming tool should also include a "software generator". The software development process is accelerated by making the machine write the code for the graphical representation of the code. Software developers should not have to worry about the details of the code. Instead, they should focus on finding the optimum solution to a problem. It is impossible for people to create bug free programs due to human errors. Therefore, it makes sense to try to automate the software creation process as much as possible. Automating the process eliminates some of the bugs that might have been introduced otherwise.

This thesis introduces a graphical editing framework for the StreamIt language. Better programming tools are required to accelerate the software development process in the DSP domain. As a result, a prototype graphical editor has been developed for the rapid creation of StreamIt programs. This tool will allow developers to work more efficiently by automating certain processes and providing intuitive visualization tools.

Section 1.1 provides an overview of the StreamIt Graph Editor. Section 1.2 discusses the organization of the rest of the thesis. Finally, the last section in this chapter (1.3) presents a motivating example of how the StreamIt Graph Editor simplifies the development process for the developer of stream programs.

1.1 Overview

A programming language can be more useful if it provides the right level of abstraction. Not only does the right level of abstraction make programming more intuitive, but it also allows for the creation of tools that take advantage of the language's internal representation. Such a programming tool can make the process to develop software easier and more straightforward. As a result, an increase in productivity can be expected when these advantages are exploited.

StreamIt is a programming language and compilation infrastructure designed to facilitate the programming of streaming applications while at the same time maintain efficiency [11]. The StreamIt language has a hierarchical structure that can be best expressed with a graphical environment that can capture and represent the stream

graph structure. The stream graph structure itself provides many opportunities for good visualization tools. A text editor does not have all the functionality needed in the ideal development environment for the creation of StreamIt programs.

The StreamIt Graph Editor (SGE) is a graphical tool built as a plugin for Eclipse, a universal programming tool from IBM [3] that provides a simple and intuitive programming interface that takes advantage of StreamIt's graph structure. Although execution of a StreamIt program follows a complex concurrent pattern, the streaming nature can lead to simple visualization of the current state of the program.

The SGE can improve developer performance by serving both as a visualization tool and as an automatic engine for creating StreamIt programs. The increase in performance results from providing a method to do things faster and reduce human errors.

A visual representation of StreamIt code makes sense due to the nature of the language. The advantage of having this graphical representation is that the developer can see the components on the screen instead of having to mentally visualize them. This is especially true for large, complex graphs. The developer might get lost in the code trying to figure which components are connected to each other by having to go through pages of code.

Allowing the SGE to automatically create code for the user minimizes the number of possible errors since SGE will automatically create the code according to the current state of the connections and the properties of the components in the graphical editor. In other cases, the SGE also minimizes errors when trying to refactor components. Refactoring is the process of combining two or more components into a single component. It is more likely that the user might make a mistake while manually doing all the steps to refactor the components than if the SGE does this automatically.

The SGE provides all of the features expected from a graphical editor so that the user can fully take advantage of the tool. Some of the features in the SGE include automatic layout of the graph, zoom in/out capabilities, expanding/collapsing of StreamIt hierarchical nodes, the modification of properties of specific stream structures, and the ability to copy/paste StreamIt object.

1.2 Organization

The remainder of this chapter will present a motivating example for the usefulness of a programming tool such as the StreamIt Graph Editor (1.3). Chapter 2 presents background information on the StreamIt language (2.1) and mentions related work (2.2). Chapter 3 contains a description of the features one would expect in the StreamIt Graph Editor. A prototype version containing some of these features was developed. Chapter 4 discusses the prototype version of the SGE and possibilities for future work. Chapter 5 presents the concluding remarks.

1.3 Motivating Example

The StreamIt Graph Editor has several advantages over the typical way of developing programs by using a text editor. The following example will illustrate how the SGE simplifies the process of creating StreamIt code by automating functionality and providing a graphical view of the stream program. Furthermore, it will be shown that the StreamIt Graph Editor reduces the possibility of any unexpected errors.

The StreamIt language was developed to facilitate the development of streaming applications. The basic building blocks in the StreamIt language are filters. Each filter inputs some data, processes the data and outputs new data. StreamIt has certain constructs that can be used as containers in different hierarchy levels: pipelines, splitjoins and feedback loops. The example in this section deals with splitjoins. A splitjoin contains parallel streams encapsulated within a splitter (one input, multiple outputs) and joiner (multiple inputs, one output).

Assume that we have some StreamIt code in which there are two splitjoins (A and B) that we want to merge. The result of merging two splitjoins is a new splitjoin (C) that contains the inner children of the two splitjoins (A and B). Figure 1-1 shows the result of the merge just described.

In addition to the merge, another change will be to duplicate the last inner child of splitjoin B thirty times. The final result of the merge and the duplication will

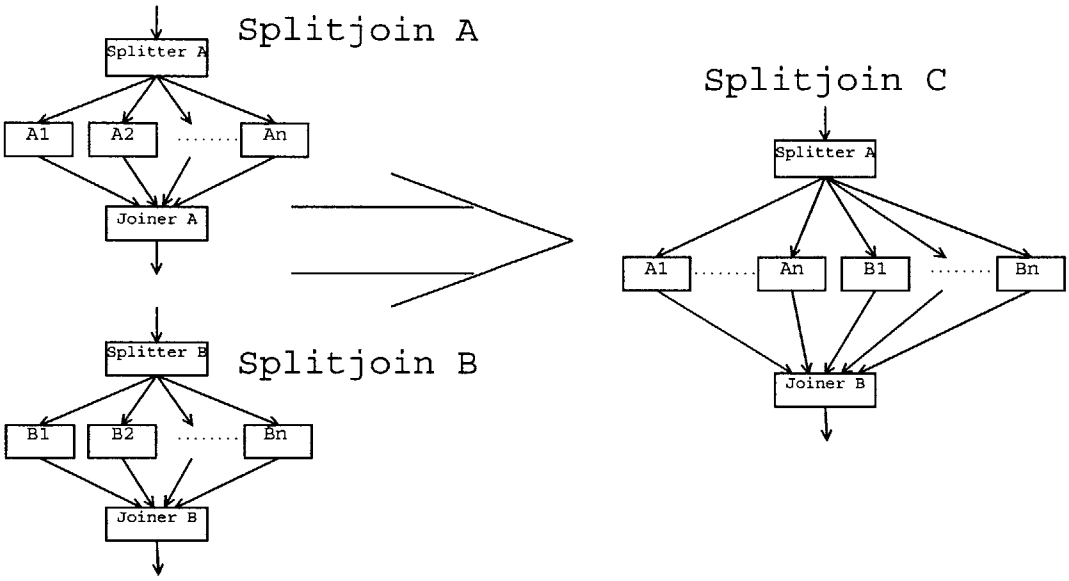


Figure 1-1: Merging splitjoin A and splitjoin B to produce splitjoin C

be a splitjoin with the following inner child structure: inner children of splitjoin A + thirty instances of the last inner child of splitjoin B + the rest of the children of splitjoin B.

Let us first examine the process that someone using a text editor has to go through in order to merge the two splitjoins. The first step is to identify where the code belonging to splitjoin A and splitjoin B is located. There is the possibility that the code corresponding to each of these two splitjoins will not be adjacent. As a consequence, the user will have to constantly move around different sections of the code to perform the necessary actions. This causes problems since it increases the chances that the user might copy or paste the code in the wrong location when trying to manually merge.

In addition to problems related to human error, it also takes a significant amount of steps to manually perform the merge and the duplication. Once the user has identified the two splitjoins to merge, the code for splitjoin C has to be created. The code that adds the inner children and the splitter of splitjoin A must be copied in the declaration of splitjoin C. The same has to be done with the inner children and the

joiner of splitjoin B. The user has to be careful that the elements have been copied in the correct order. The references to splitjoin A and splitjoin B can now be removed (the places in the code where these two splitjoins were added). An alternative to the process above would have been to modify splitjoin A to be the new splitjoin C. Although this saves a couple of steps, the user still has to perform a considerable number of actions. After the merge, the user would proceed with the duplication of the last child of splitjoin B. The add statement for this child must be selected and duplicated the correct number of times (perhaps using a for-loop). The user is done once this last step is completed. However, one should note all the steps that were required. The large quantity of steps increases the probability that a mistake was made at some point.

Now let us analyze how the StreamIt Graph Editor can be used to achieve the same objective described above. First, the user will have no problems identifying the splitjoins that have to be merged since we are dealing with a graphical representation of the stream graph. The zoom option can be used to adjust the view accordingly. In the same manner, the user can expand or collapse structures to adjust the level of detail so that only the relevant blocks are shown. These tools are provided in order to enhance the developer's efficiency by making things easier.

Once the two splitjoins that will be merged have been selected, the automatic merge option can be selected. The SGE will handle all the steps that had to be done manually in the case where the text editor was used to modify the source code. The user can now select the child to duplicate in the new splitjoin and choose the duplication option in the SGE. The number of times the child will be duplicated must be specified. The SGE will make the corresponding changes to the graph.

The user can now continue making modifications to the graph representation of the stream program. Once the user is done with all the changes, the option to generate the code for the graph can be selected.

The example above was a simple one. More complicated cases would even take more advantage of the SGE's functionality. Manually merging the inner components of the splitjoin as well as the splitjoins themselves is prone to more human errors.

Chapter 2

Background

StreamIt is the programming language for which the graphical programming environment described in this thesis was designed for. Section 2.1 presents an overview of StreamIt language. The details of the StreamIt language are discussed in Section 2.1.1. The last Section of Chapter 2 describes related work in the area of programming (prototyping) tools for the development of DSP applications.

2.1 StreamIt

The StreamIt language [11] [10] was developed at the MIT Computer Science and Artificial Intelligence Laboratory to facilitate the development of streaming applications such as digital signal processing. Each data item in a streaming application is in the system for only a small amount of time. Stream programs also have regular communication patterns and abundant parallelism. In StreamIt, these properties are exposed to the compiler while maintaining a high level of abstraction for the developer. StreamIt is intended to simplify coding of DSP and other streaming computations.

Languages such as C, C++ and Matlab have been traditionally used for developing DSP applications. However, programming streaming applications in these languages is generally tedious and error prone since development is done at lower abstraction levels. Unlike the scientific domain, stream programs are characterized by regular communication patterns, abundant parallelism, and short data lifetimes. All these

can be exploited to improve programmability and performance. The advantage of StreamIt is that it provides the right level of abstraction for stream programming while at the same time providing compiler level optimizations.

At this time when the streaming application domain is becoming increasingly prevalent and widespread due to the popularity of the internet and wireless communication, it is important to provide a tool that will enhance the development process of stream programs. StreamIt is a language that can provide this functionality since it is easy to use and efficient at the same time.

2.1.1 Language Specifics

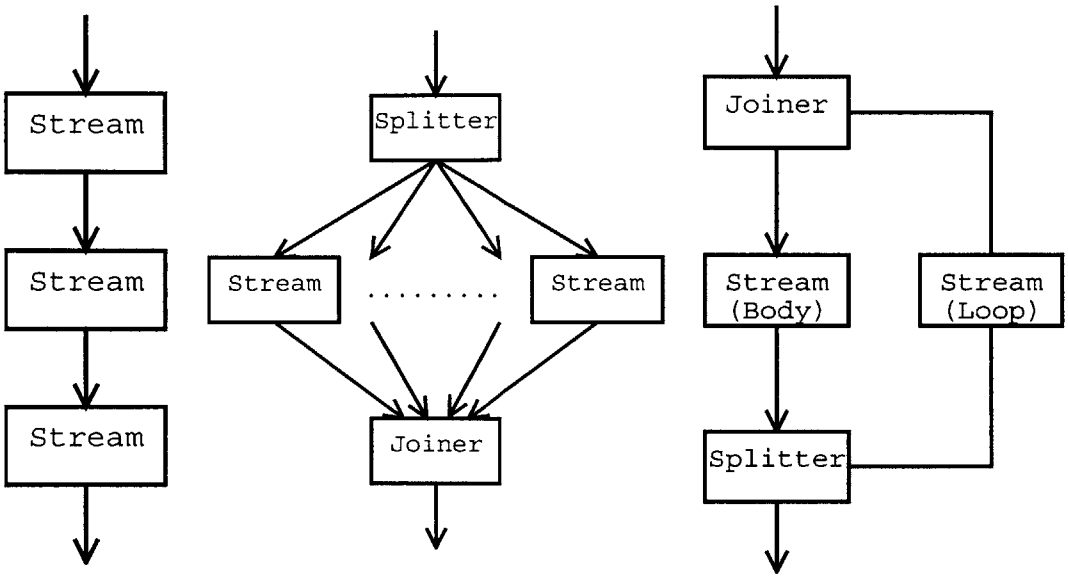
The StreamIt language provides a simple high level abstraction that allows software engineers without lower level DSP knowledge to easily represent stream programs. Perhaps the most useful abstraction that StreamIt provides is the categorization of streaming programs [10].

The basic building block in the StreamIt language is a filter. Each filter inputs some data, processes the data and outputs the new data. The values that the filter reads are taken from its input tape (pop) and the values it writes are placed on the output tape (push). A filter also has a work function which describes the filter's atomic execution step.

Programs in the StreamIt language can be described as hierarchical graphs of filters. This representation is useful because streaming algorithms are also viewed the same way from a digital signal processing perspective. The constructs that can be used as containers to hold filters in different hierarchy levels are pipelines, splitjoins, and feedback loops. Every subcomponent of a structure is a stream. Figure 2-1 shows the container structures in the StreamIt language.

A pipeline contains streams that are arranged sequentially one after another. The output of one stream will be the input of another stream in the pipeline. The top level pipeline is the outermost pipeline that contains the rest of the streams in the graph.

Splitjoins, on the other hand, contain parallel streams that are encapsulated within



(a) A pipeline. **(b) A splitjoin.** **(c) A feedbackloop.**

Figure 2-1: Container structures supported by StreamIt.

a splitter and a joiner. The way in which data is taken in and out of a splitter or joiner varies depending on the type. A splitter can either be a duplicate or a roundrobin. A duplicate sends copies of the data to all its outputs. On the other hand, a roundrobin distributes items cyclically according to its array of weights (w_0, w_1,..., w_n). The roundrobin will send w_0 items to the first child, w_1 items to the second child, and so on until the last child receives w_n elements. Splitters and joiners can support any number of child streams.

Feedback loops allow cycles to be introduced into the stream graph. A feedback loop is composed of a joiner, a body, a splitter and a loop. The joiner takes in an input from another node and connects to the body. The body is connected to the splitter which in turn connects to an element outside the feedback loop and to the loop of the body. The loop takes inputs from the splitter and sends its output to the joiner.

The hierarchical structure of StreamIt can be shown in Figure 2-2. The encapsulating component of filters F1 and F2 is pipeline B. Splitjoin C is the encapsulating

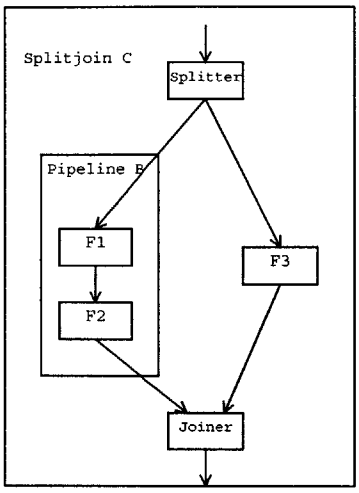


Figure 2-2: Hierarchical nature of components

node of the splitter, joiner, pipeline B and filter F3. The same hierarchical structure continues outward until the top level node is reached.

The general process of creating an application in StreamIt can be described as follows. The programmer constructs a stream graph containing blocks with a single input and a single output, and specifies the functionality of each block. Then, the programmer has to define the structure of the container structures (the way in which the blocks are connected to each other). The compiler will generate code for each block, and at the same time apply optimizations to the stream graph to produce efficient code. StreamIt allows the easier development and debugging of stream programs since it resembles a higher level programming language.

Figure A-1 contains the source code for a StreamIt program that simulates how echos are added to sound waves. The sound waves are represented as digital data. The original sequence of data will be added to a time-shifted version of the original sequence. The toplevel pipeline of the program is EchoEffect2. This pipeline contains all of the other constructs in the stream graph. A number is generated by IntSource and it is written to its output tape. The input tape of Echo (a feedback loop) receives the value from IntSource. Eventually Echo will produce an output and put it on Adder's input tape. Adder takes the top two values in its input tape and adds

them. The sum will be printed by IntPrinter.

2.2 Related Work

The Generic Modeling Environment [6] is a configurable toolkit for creating domain specific modeling and program synthesis environments. This modeling tool also uses Eclipse as its IDE. The GME is more of a generic modeling environment rather than a development environment specific to DSP applications like the SGE.

Matlab's Simulink [7] is an object oriented dynamic simulation package for modeling, simulating, and analyzing dynamic, multidomain systems. Simulink is usually used for control system design, DSP system design and other simulation applications. Although both Simulink and the SGE provide graphical tools for the DSP domain, Simulink's main goal is to provide a simulation environment while SGE's goal is to provide a programming environment.

Texas Instrument's Research and Development engineers are working on a DSP prototyping tool that uses MATLAB, Simulink, the Signal Processing Blockset, and Real-Time Workshop along with TI development tools [8]. This tool will use interactive block-diagram simulation and automatic code generation. The goal is to allow DSP engineers to refine implementation details directly in the system model and produce real-time software prototypes without traditional programming. Both TI's tool and the SGE share the objective of becoming a development environment that increases programmer efficiency. The SGE is more simple in that it is not a combination of other tools and that it tries to take advantage of StreamIt's specific constructs.

LabVIEW [5] provides a powerful graphical development environment for signal acquisition, measurement analysis, and data presentation. LabVIEW is similar to SGE in that it tries to simplify things for the developer by providing a graphical development environment. However, LabVIEW is mainly used for data acquisition rather than actual program development.

Chapter 3

A Graphical Editor for StreamIt

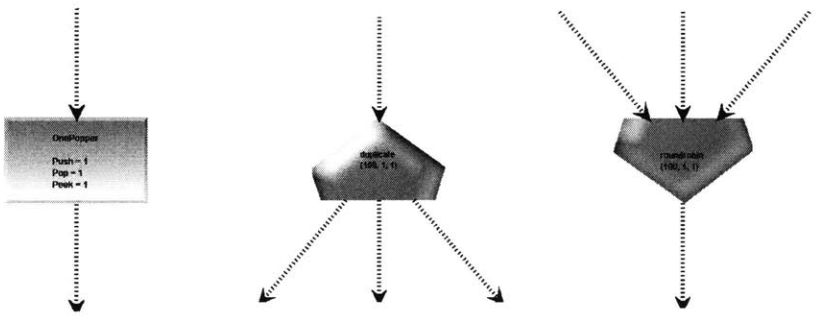
A graphical editor for StreamIt must provide all the functionality required to accelerate the development of DSP applications. At the same time it must provide all (or most) of the functionality needed to create StreamIt programs independently from the text editor. This chapter describes the most important features and the properties for such an infrastructure.

3.1 Intuitive Visualization of Components

The graphical editor makes it possible to create a graphical representation of legal StreamIt code. A graphical representation of the stream program makes it easier for the developer of the application to see how everything fits together as a whole without having to visualize everything on his mind. This is especially useful for large complicated graphs. The visualization element allows the developer to view the overall design of the program. Any flaws in the design would be easier to detect having the graphical representation instead of just looking at lines of code.

It is important that the elements in the StreamIt Graph Editor are intuitively represented on the screen. The user should be able to clearly distinguish a filter from a splitter or a pipeline. Therefore, the components should be represented in an intuitive manner so that they can be easily identified.

One such representation of the elements is the following. Directed arrows connect-



(a) Filter. **(b) Splitter.** **(c) Joiner.**

Figure 3-1: Component representation in SGE.

ing one component to another can illustrate the flow of data in the graph. Filters are like "black boxes" whose internal function must be specified by the programmer. A filter can be represented as a solid box since they represent the basic building block of StreamIt (Figure 3-1 (a)).

The shape of a splitter should be something that clearly illustrates that it is a structure that takes one input, but may have several outputs. An inverted funnel-like figure would serve such a function (Figure 3-1(b)). Similarly, a joiner could have funnel-like shape to show that it supports several inputs, but only one output Figure 3-1(c)).

Container structures (splitjoins, pipeline, feedback loops) also require an intuitive representation. The simplest way to represent these containers would be to have rectangular borders that encapsulate the objects within them. Pipelines, splitjoins and feedback loops would each have a different color assigned to them in order to distinguish them from the other types of containers.

Different shades of color could be used for all of the StreamIt components to distinguish the hierarchy level in which they are located. Components that have the toplevel pipeline as their immediate encapsulating node can be shaded lightly. The more embedded components are within other encapsulating nodes, the darker the shade of color that would be assigned to them.

Figure A-1 contains the source code for a StreamIt program that simulates how

echos are added to sound waves. Figure B-1 contains the graphical representation of the StreamIt program. Notice how it is easier to view the overall design and flow of data in the graph. The first component in the graph is IntSource. The output of IntSource is connected to the feedback loop. The graph clearly shows the way that the components in the feedback loop. The output of the feedback loop is then connected to the Adder filter that gives its result to IntPrinter. Without the graph, the user would have to spend some time trying to figure the graph's structure. This process would be time consuming in applications that have many container components embedded in each other.

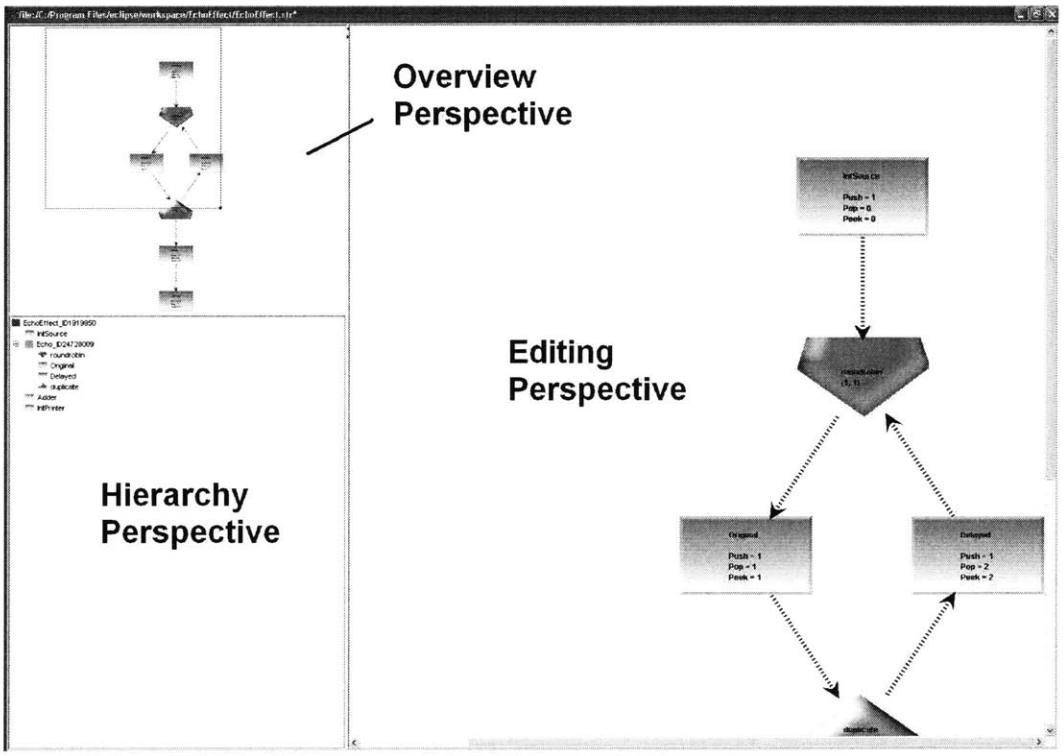


Figure 3-2: Different perspectives of the stream graph

3.2 Alternate Perspectives

The SGE must provide the user with different perspectives of the model. The perspective most appropriate for a situation can then be used. Some actions can be performed faster using a certain perspective. In addition, having the different perspectives can also provide more information to the user at any given moment. Three perspectives essential to a graphical editor include editing, overview and hierarchy. Figure 3-2 shows how these perspectives fit together.

3.2.1 Editing Perspective

The main perspective where most actions are performed is the editing perspective. The developer can make graphical changes to the stream graph representation of the code using this perspective. Changes that occur in this perspective should immediately be reflected in the other perspectives. Some of these changes include addition of nodes, deletion of nodes, and reordering of nodes.

3.2.2 Overview Perspective

The overview perspective provides a way in which the user can examine the entire graph. The user can see the entire stream graph structure even when dealing with large, complex graphs. A bounding box in the overview window would allow the user to identify the region of the graph that is currently being shown in the editing perspective. The current region in the editing perspective changes when the bounding box is moved around the overview window. The box can also be resized in order to zoom in or zoom out of the graph.

3.2.3 Hierarchy Perspective

The hierarchy perspective shows the hierarchy of the stream graph. The hierarchical nature of stream graphs makes it possible to have nodes encapsulated by other levels of nodes. A tree model can be used to quickly examine the hierarchy of the graph

(Figure 3-3). Modifications of the graph should be allowed from this perspective. The user could select a component in the hierarchy perspective, press the Delete button and the change would be reflected on all of the other perspectives.

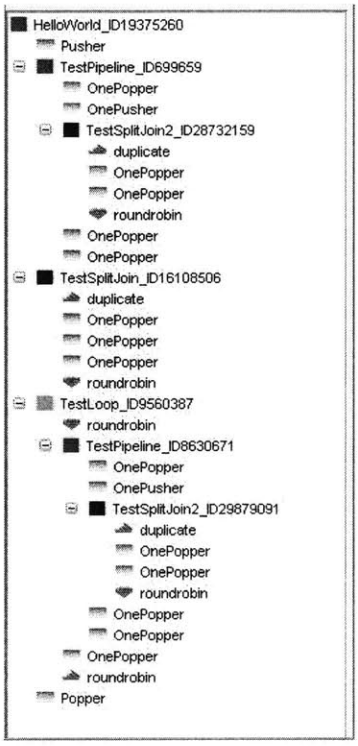


Figure 3-3: Hierarchy Perspective

3.3 Graphical Editing

One of the essential features of the SGE is that the developer should be able to graphically edit the StreamIt graph and hence the program. The SGE must allow the addition, deletion, and connecting of StreamIt components.

Instead of writing source code, the programmer selects the type of component that needs to be inserted in the stream graph. The StreamIt components that can be added this way include filters, splitters and joiners. Container structures (pipelines, splitjoins and feedback loops) can be added to the graph only when elements are

grouped together. All the grouped components will have the same new container encapsulating them. Graph editing seems more intuitive due to the hierarchical nature of StreamIt.

It also seems natural to allow the user to specify connections between the different components in a graphical manner. Connecting an edge from component A to component B represents that the output from A is connected to the input of B. The process of making the connections can be compared to the one of specifying the data flow in the graph.

3.4 Property Modification

The user should be able to modify the properties of each of the elements in the graph. The main properties that can be modified for any node include the name, the encapsulating container, and the input/output types. Other elements such as a splitters and joiners can have their weights set. Filters can have their push, pop and peek values modified. Property modification should be possible from the moment that a component is created.

The advantage of allowing the SGE to control the modification of component properties is that it can prevent the user from entering illegal property values. The same error would not be detected in the text editor until the user has compiled the program. An example of such an error is having components with the wrong input/output types connected to each other. All changes that result in illegal configurations of the graph should be prevented as soon as possible.

The StreamIt application developer should also be able to change the properties for several selected components at the same time. The SGE should determine the common properties that can be modified for the selected components. Just as if the SGE were dealing with a single structure, any property changes that cause problems are disallowed.

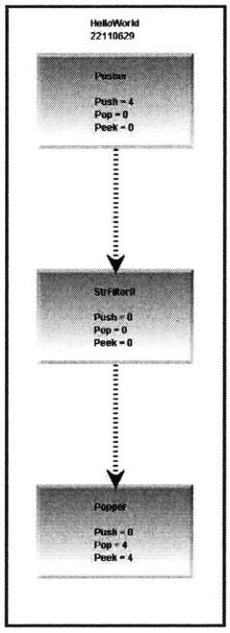


Figure 3-4: Pipeline Layout

3.5 Graph Layout

The stream graph has to be graphically laid out in an intuitive manner. The SGE should automatically order the components in a container according to the connections among them. The layout should prevent the cluttering of the different components by allowing enough space for the user to make modifications. In addition, the layout of the stream graph must maintain the hierarchical nature of the graph. The user should be able to easily identify the different types of components by their arrangement. The elements of a pipeline are arranged in a sequential manner (Figure 3-4). A splitjoin is laid out with its splitter at the top, all the children evenly distributed in the middle and its joiner at the bottom (Figure 3-5). The components inside a feedback loop are arranged in a cyclic manner (Figure 3-6). Note how each of the container structures encapsulate the streams within them.

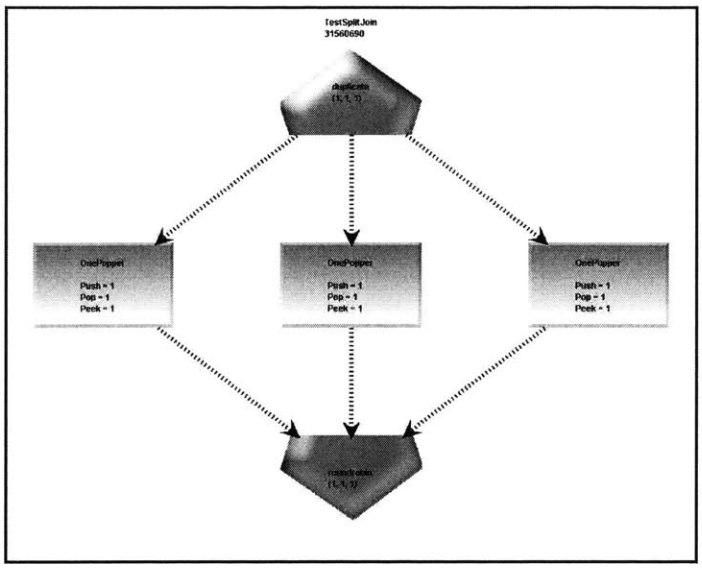


Figure 3-5: Splitjoin Layout

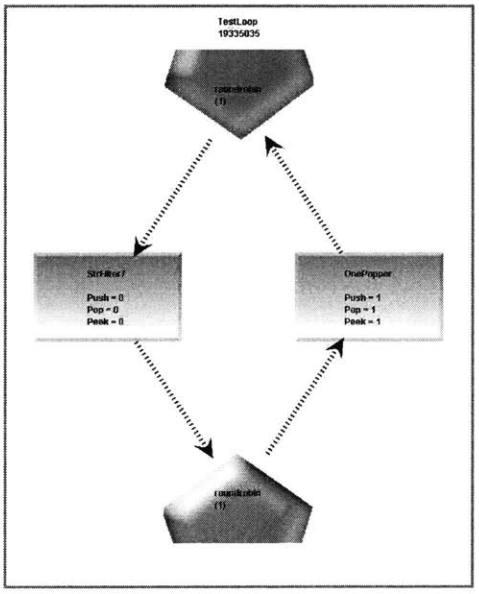


Figure 3-6: Feedback Loop Layout

3.6 Display Tools

A graphical editor needs to provide the user with tools that change how objects are displayed on the screen. These tools are essential when editing complicated structures.

They improve performance by allowing the user to focus only on the relevant parts of the graph that must be examined or modified.

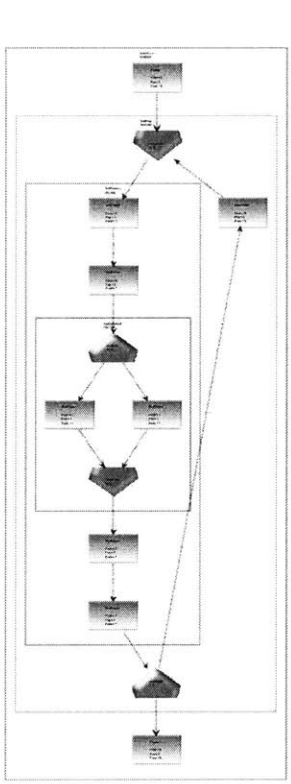
The zoom functionality is extremely useful when the entire graph does not fit on the screen. Too much time would be wasted if one had to change to another section of the graph manually by moving the scroll bars. Zooming accelerates this process since the user can quickly identify the region of the stream graph that has to be examined. The zoom capability permits all the necessary components to fit on the window.

Collapsing and expanding of container nodes allow control over the level of detail visible to the user. The overall structure of the graph remains intact, but fewer components are shown. Assume that a graph contains a pipeline with many internal elements and this pipeline does not have to be modified anymore. It would be easier to work on the rest of the structure by treating this complex pipeline as a "black box" and visually collapsing the structure. Thus, the developer can concentrate on the interactions between components rather than on details. Figure 3-7(a)-(c) shows the effect of collapsing a graph to simplify the overall structure.

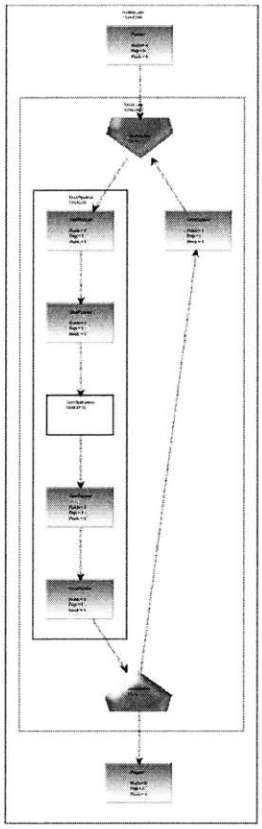
Visibility of containers is another feature that would allow the user to control how the graph is visualized. Container nodes can either be visible on the screen or they can be hidden. When container nodes are visible, the user can observe the hierarchy that exists among the different components. However, the user might choose to hide the containers in order to facilitate the editing of the graph. Figure 3-7(d) shows a graph with its containers invisible. The same graph with the containers visible is in Figure 3-7(c).

3.7 Editing Tools

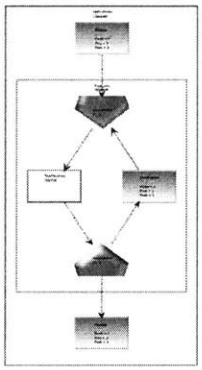
A graphical editor must have editing functionality such as copy, paste and undo. Copy/paste increases the productivity of a programmer by allowing the reuse of existing structures. The stream graph might contain a component that needs to be used again in a different part of the graph. Instead of creating a node with the same properties, the user can just copy the object.



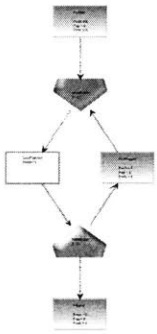
(a) **Fully expanded.**



(b) **1 level collapsed.**



(c) **2 levels collapsed.**



(d) **Containers Invisible.**

Figure 3-7: Expanding/Collapsing Components

A special paste functionality that would allow several instances of the copied object to be created would be useful. A specific case where the special copy/paste functionality would be useful is in the duplication of the inner nodes of a splitjoin. Automating the creation of multiple instances of a splitjoin's inner node would speed up the development process for the programmer. Creating multiple instances of the node would only require the programmer to specify how many copies must be created.

The ability to undo changes helps the user correct mistakes. Many times the action performed was an error or had undesired result. Undoing the changes can bring the user back to a state of the stream graph without the problems that had been introduced.

3.8 Merging

Merging is another action that takes advantage of the automating capabilities of the graph editor. Merging allows different container nodes to be selected in order to create a single new component from the inner components of these structures. The result of merging two splitjoins (A and B) is a new splitjoin (C) that contains the inner children of the two splitjoins (A and B). Figure 1-1 shows the result of merging two splitjoins.

Merging simplifies things for the user by automating the process. If this action was to be done manually by the user, it would take more steps and it would also make it more likely for an error to occur. The SGE should also automatically determine when it is possible to merge components into a single component according to the structure of the graph.

3.9 Image Export

The SGE should export images of the graph that is being edited. Different formats should be supported. Having an image available can allow the programmer to further analyze the design of the application being built. It is easier to explain the function-

ality of a StreamIt program if in addition to the code, the stream graph image is also provided.

3.10 Code Creation

Ideally, there should be "isomorphism" between the graphical representation of the stream graph and the source code. If the source code is modified, then the change should be reflected in the graph. On the other hand, if the graph is modified, the change should be reflected in the source code. This would allow the user to switch between the graphical editor and the text editor at any point. To start developing an application, the programmer can use the graph editor for visualization purposes. Once the programmer is done creating the overall design of the application, the text editor can be used to fill in the details.

Chapter 4

Implementation of the StreamIt Graphical Editor

A prototype of the StreamIt Graph Editor was developed. The SGE was designed as a plug-in for Eclipse, a universal tool platform from IBM. The first section (4.1) of this chapter will describe relevant components in the design of the SGE. Section 4.2 presents some of the features and limitations of the SGE. Finally, Section 4.3 discusses future work.

4.1 General Overview

4.1.1 Eclipse

The StreamIt Graph Editor was built as a plug-in for IBMs Eclipse [3]. Eclipse is an extensible integrated development environment (IDE). The Eclipse platform has a mechanism for finding, integrating and running plug-ins. When Eclipse is launched, an IDE composed of a set of available plug-ins is made available to the user. The SGE has been integrated into Eclipse via in-process invocation [2]. The SGE runs in the same JVM and can use the same class libraries as Eclipse [9]. The Eclipse plug-in that launches the StreamIt Graph Editor is also responsible for resource synchronization between Eclipse and the editor.

4.1.2 JGraph

JGraph is the graph component that was used to create the SGEs visual elements. JGraph is one of the most powerful, lightweight and feature-rich opensource graph components available for Java [1]. Instead of having to program the low level interactions using Swing/AWT, JGraph provides a high level abstraction that facilitates the creation of the visual representation of graphs.

4.1.3 SGE Internal Representation

The underlying internal representation (IR) of the stream graph contains all the information necessary to display important details. The IR contains all of the nodes in the graph and their connections. The IR consists of nodes that represent any of the available structures that the StreamIt language supports: filters, pipelines, splitter, joiners, feedback loops and splitjoins. A GEStreamNode is the basic unit in the IR that represents a node and contains all of the data that is particular to that object. A GEStreamNode provides functionality to access or modify properties that are present in all of the StreamIt structures (i.e. name, encapsulating container, type). Specific structures extend GEStreamNode to provide features particular to that type of node. For example, a GEFilter contains the push, pop, and peek rates for their work function while GESplitters and GEJoiners contain roundrobin weights.

Pipelines, splitjoins and feedback loops are also GEContainer nodes in addition to being GEStreamNode due to their ability to encapsulate other components. Every node, except for the top level node has a GEContainer as its encapsulating structure. By having a GEContainer, more features in the SGE can take advantage of StreamIt's hierarchical nature.

4.2 Features

The following subsections will describe some of the tasks that can be done using the prototype version of the SGE. In addition, the limitations of the prototype will also

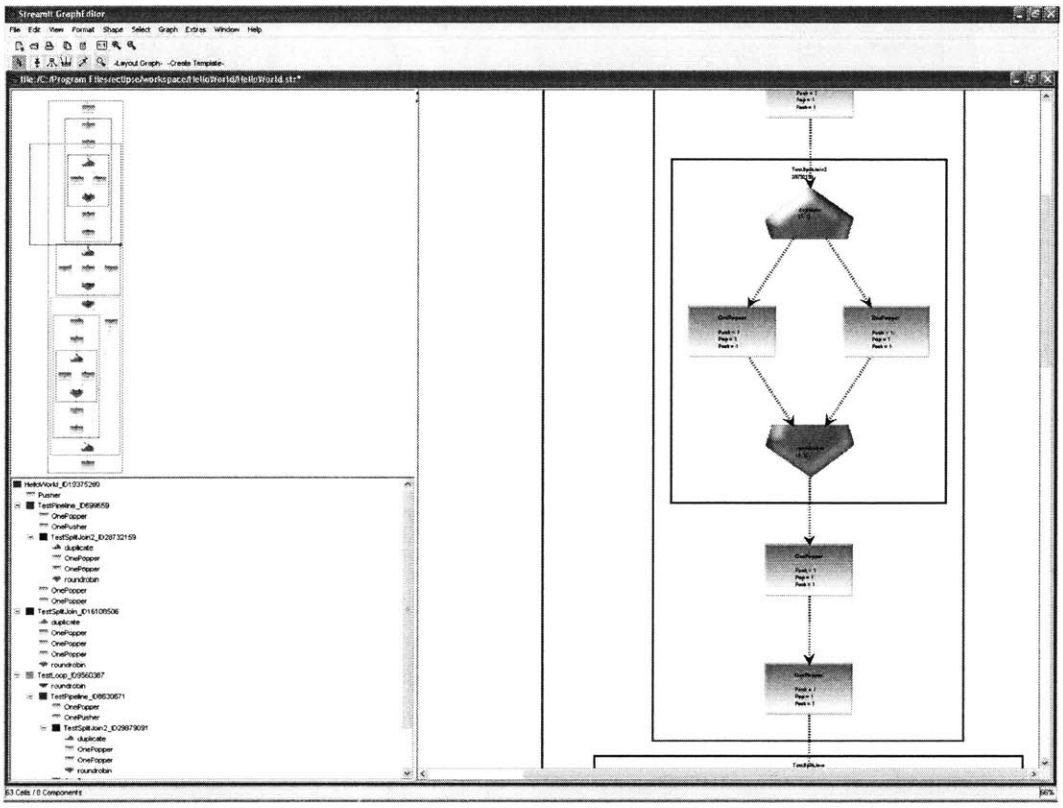


Figure 4-1: SGE Prototype

be exposed. Figure 4-1 shows the SGE.

4.2.1 Creating StreamIt components

Filters, splitters and joiners can be created by clicking on their corresponding icons on the toolbar in the editor. Figure 4-2 shows the SGE's toolbar. Once the type of the component that one wants to add has been selected, the user can use the mouse to create the bounding box where the component is to be added.

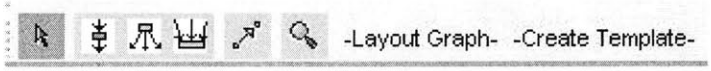


Figure 4-2: Toolbar

Splitjoins, pipelines and feedback loops are created a different way. The com-

ponents that will be part of the container structure must first be selected so that they can be "grouped". The effect of grouping is that all of the selected structures will have the same new encapsulating container. This process is similar to grouping of elements in PowerPoint in that an association between the selected components is created. Each of the container nodes will have a unique ID so that there is no ambiguity at the time that an element is to be added to a container.

In the current version of the SGE prototype, there are no restrictions for grouping elements into a pipeline. However, if one wants to group into a splitjoin or a feedback loop, a splitter and a joiner must be among the elements selected to group. Splitjoins and feedback loops are not legal if they do not include these components. Therefore, it does not make sense to allow the creation of stream graph components that are incomplete.

4.2.2 Connecting Components

Graph components can be connected graphically. Once the user has selected the edge connection option from the editor's toolbar, the source component must be selected and the created edge must be dragged to the target component. Not all connections are allowed since they would result in illegal stream graphs. The SGE will warn the user whenever a connection is not possible (i.e. connecting an element to itself).

Connections are deleted the same way as other components of the graph. Certain components such as filters require existing connections to be deleted before allowing any more connections. Filters can only have one input and one output so trying to create connections that would result in more than one input/output is not allowed.

Component connections determine the ordering of elements within their encapsulating container. Elements belonging to a pipeline are automatically arranged according to the connections among these inner components. As a result, the programmer does not have to worry about specifying which element is the first one in the pipeline, which one is the second one and so on. The SGE does this automatically when the user connects the different structures.

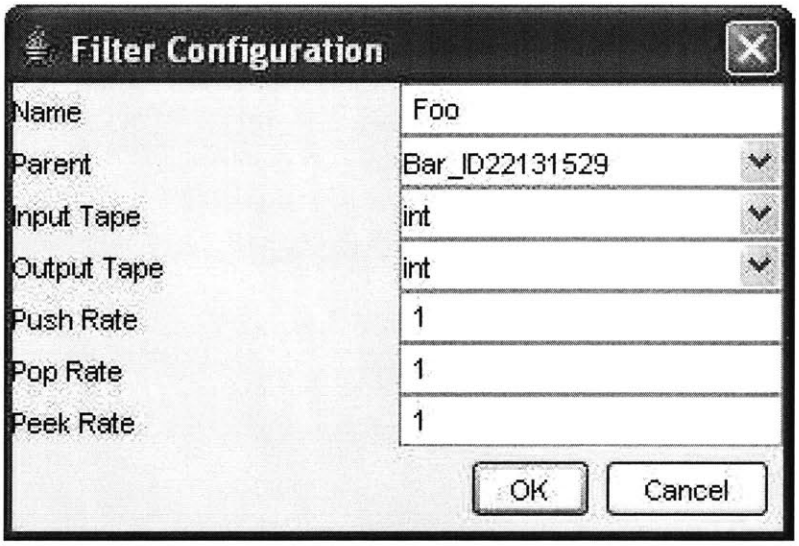


Figure 4-3: Property Window

4.2.3 Changing Properties

The properties of a component can be modified using the SGE. The developer selects the desired element and then selects the properties option in the menu or by right-clicking. A window containing the current values of the component's properties will appear on the screen.

The specific properties that can be modified depend on the type of the component selected. Splitters and joiners have weights, but they do not have an input type or an output type. Their input and output types are determined by their encapsulating container. In addition, splitters do not have a name field. Instead the user must specify the type of the splitter or joiner (roundrobin or duplicate).

The properties window will prevent the user from certain illegal values that might result in an illegal stream graph. The encapsulating component of a container cannot be itself or any other element that it contains. In addition, an element's encapsulating container cannot be changed if the node selected is connected. Changes like this could cause the hierarchy of the graph structure to be nonsensical.

In stream graphs, it is commonly the case that there is more than one instance of the same component in the graph. These instances share the same "class" name and

the same properties. If the properties of one of these instances are modified, then the name of the modifed component must also change because it no longer shares the same properties as the other instances. The SGE changes the name of the structure automatically so that when the code for the stream graph is generated, the definition for the new type will be included. Ideally, the SGE should have prompted the user to perform the action described above or to actually change the properties of all the nodes with the same "class" name as the modified node.

The current version of the SGE does not support the modification of the properties of multiple elements. Properties must be modified individually. When changing the properties of multiple components, it is more likely that illegal configurations of the graph are created. For example, selecting all of the elements in the graph and changing the encapsulating container to be the same for all of them is impossible. The SGE would have to consider if any problems are caused by the desired modification before the change takes effect.

4.2.4 Expanding/Collapsing

Expanding and collapsing of stream nodes is done according to the hierarchy level in which nodes are located. Collapsing and expanding occurs one level at a time when no nodes are selected in the editing window. The nodes at the deepest level are the ones that have more ancestors between them and the toplevel pipeline. If the graph is fully expanded, the deepest nodes will be collapsed first. Next time the graph is collapsed, the next deepest nodes will be collapsed.

Containers can be expanded or collapsed by selecting the expand or collapse option from the menus. Specific containers can be collapsed by explicitly selecting them and then clicking the collapse button. Instead of collapsing all of the containers at a certain level, only the selected component will be collapsed.

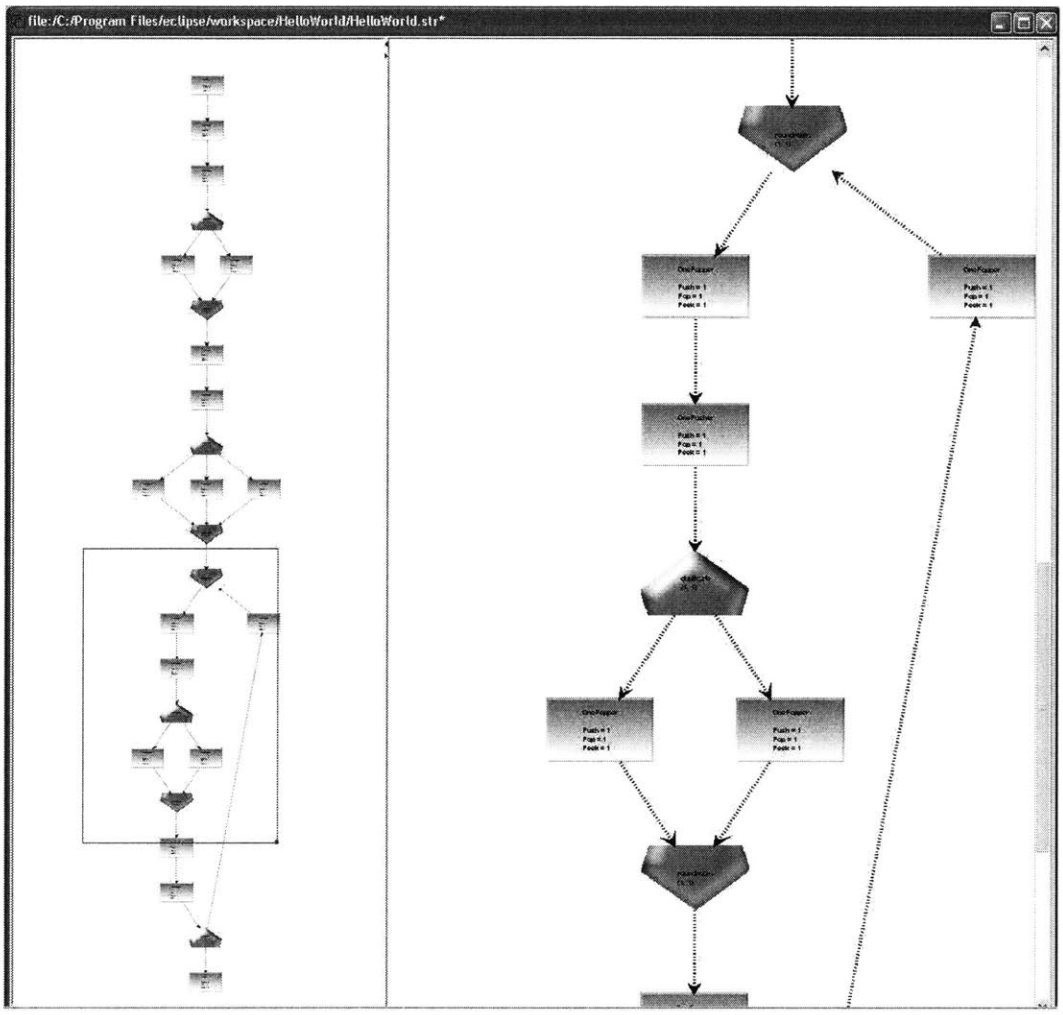
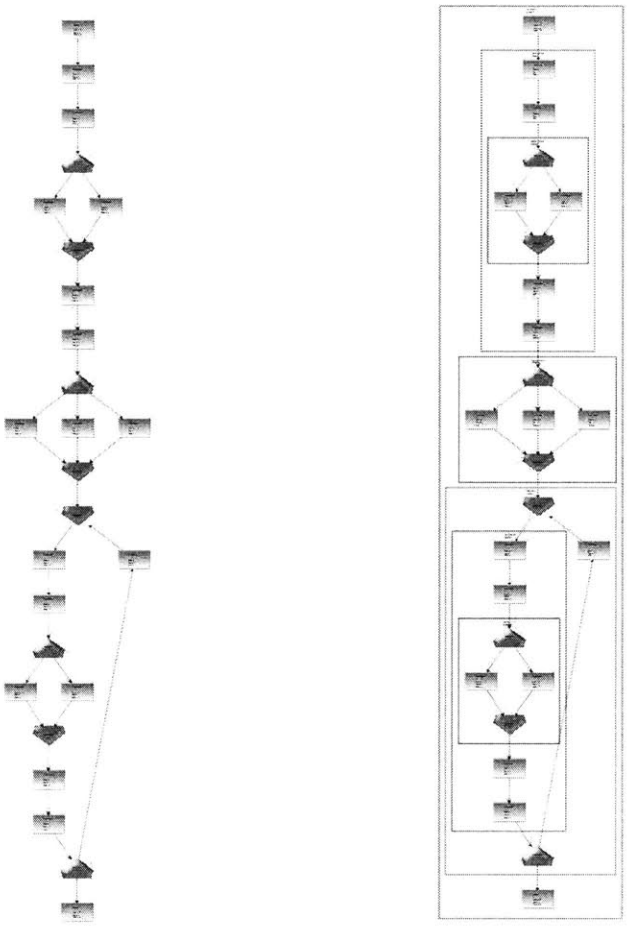


Figure 4-4: Zoom (using hierarchy perspective)

4.2.5 Zoom

The SGE provides functionality that allows the user to specify how much of the graph should be visible in the editing window. Zooming can be achieved by selecting the appropriate option from the menus. One can also zoom by changing the size of the bounding box in the hierarchy panel. To zoom in, the bounding box is made smaller and to zoom out the bounding box is made bigger. Other options in the view menu, allow the graph to be fitted into the window. The level of zoom can also be specified by the user.



(a) Containers invisible. **(b) Containers visible.**

Figure 4-5: Visibility of containers

4.2.6 Visibility of Containers

Containers can be made visible or invisible. It might be good to make the containers visible to analyze the hierarchy structure of the graph. However, there might be cases when the containers should be invisible in order to facilitate the editing of the rest of the graph. Containers can be made visible/invisible by simply selecting the corresponding option in the menu. Figure 4-5 shows how the same graph looks with the containers visible and invisible.

4.2.7 Copying/Pasting/Deleting

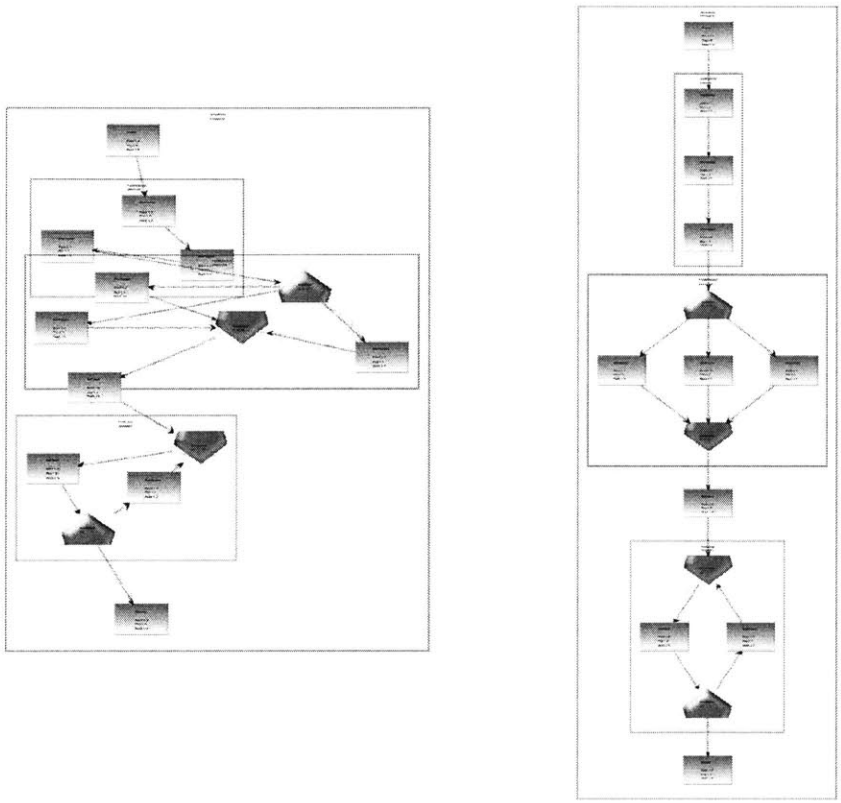
The SGE supports the basic functionality that would be expected from a graph editor. In order to copy an element, it must first be selected. Then, the copy option from the menu must be chosen. The user can now use the paste option to create a new instance of the node that was copied. It is not recommended to copy/paste large, complex containers in the prototype version of the SGE. Copying/pasting has to be optimized so that it takes a reasonable amount of time to perform the action on these containers.

The user can delete a component by pressing the Delete button or by selecting the corresponding option from the menus. If the selected component is a container, then all of its inner components will also be deleted. It is not possible to delete splitters or joiners that belong to a splitjoin or a feedback loop since these containers are not defined without their splitter or joiner. The user would never have to delete a splitter or a joiner to substitute it for another one since this is possible just by changing the component's properties.

4.2.8 Graph Layout

The SGE allows the graph to be laid out in an intuitive manner. As the stream graph is created it might become obfuscated by virtue of the programmer's actions. Adding new components, making connections, and moving components around will eventually lead to a graph with no structure. Figure 4-6(a) shows an extreme case of what might eventually happen to the graph. The problem can be solved easily by selecting the"Layout Graph" option. The end result is shown in Figure 4-6(b).

Elements that are not connected to the graph appear at the end of the container to which they belong. In future versions of the SGE, this can be modified so that unconnected components appear in another region where the user can easily identify the components that still need to be connected.



(a) Obfuscated graph. **(b) Laid out graph.**

Figure 4-6: Graph Layout

4.2.9 Merging

The SGE currently supports the merge of pipelines or splitjoins. If two splitjoins are to be merged, they must be selected from the graph first. The first splitjoin that is selected is the one that will contribute its splitter to the new splitjoin and will also have its inner components first. The second splitjoin selected will provide its joiner and its inner components will come after the components of the first selected splitjoin. When the merge option is selected, the user will be asked if the unused splitter and joiners from the selected splitjoins should be deleted. It might be helpful not to delete the old splitter and joiner to know the weight values for those components while setting the values for the new ones. When merging pipelines, the order in

which the desired pipelines are selected determines if the pipeline's elements come before or after the other pipeline's elements.

The prototype SGE only supports the merge of two components at a time. However, this functionality can easily be extended to support the merge of more components. In addition, it can also be extended to support the merge of feedback loops.

4.2.10 Specialized Splitjoin Functionality

The unique properties of the splitjoin allow the SGE to provide special functions to deal with this structure. Frequently, a splitjoin might contain several instances of the same structure as an inner component. It would be time consuming to create such a splitjoin since the user would have to copy the desired element a certain number of times and then manually have to connect the component to the splitjoin's splitter and joiner. The SGE simplifies the process by providing functionality that allows the duplication of the inner components of a splitjoin. The user only has to select the component and determine the number of copies. The SGE will automatically duplicate the component that many times, make the connections between the splitter and the joiner of the splitjoin and adjust the weights accordingly. The prototype SGE does not work well when trying to duplicate large, complex containers since some changes are required to optimize this procedure.

The inner components of a splitjoin must maintain a certain order so that the weights correspond to their respective elements. The user can specify a new value for the index of a splitjoin's inner component. The SGE will automatically adjust the weights in the splitter and the joiner. The option to set the index is also available when a component is added to the splitjoin.

4.2.11 Template Code Creation

The SGE allows the creation of template code from the structure that has been created graphically. The template code will include the type definitions of each of the components according to their properties in the SGE. Only one type declaration

```
void->void pipeline HelloWorld {
     add Pusher();
     add TestSplitJoin();
     add Popper();
     add StrFilter13();
}
void->int filter Pusher {
     init {
     }
     work push 4 {
     }
}
int->int splitjoin TestSplitJoin {
     split duplicate();
     for (int i = 0; i < 6 ; i++ {
          add OnePopper();
     }
     join roundrobin();
}
int->int filter OnePopper {
     init {
     }
     work pop 1 push 1 peek 1 {
     }
}
int->void filter Popper {
     init {
     }
     work pop 4 peek 4 {
     }
}
```

Figure 4-7: Sample Template Code.

will be required for all of the instances of a certain component in the stream graph. In addition, components will also be added to the containers they belong to.

Whenever there are more than N instances of the same component adjacent to each other in the graph, instead of adding each of these instances explicitly (i.e. having N different "add" statements) the SGE will produce code that contains a for-loop to add all of them. The value of N is currently set to four, but there could be an option that would allow the user to set this threshold value. The advantage of the for-loop code creation is that this is the kind of code that a programmer would write. It does not make sense to have N different "add" statements. Figure 4-7 shows the StreamIt code that the SGE generated from the graph shown in Figure 4-8.

The current version of the SGE does not provide much "isomorphism" between the code and the graph. Whenever one of the models is modified, the change is not reflected immediately in the other. The code needs to be compiled in order for the graph to be created. The option to create template code must be selected in the SGE to reflect the graph changes in the source code. In addition, the creation of template

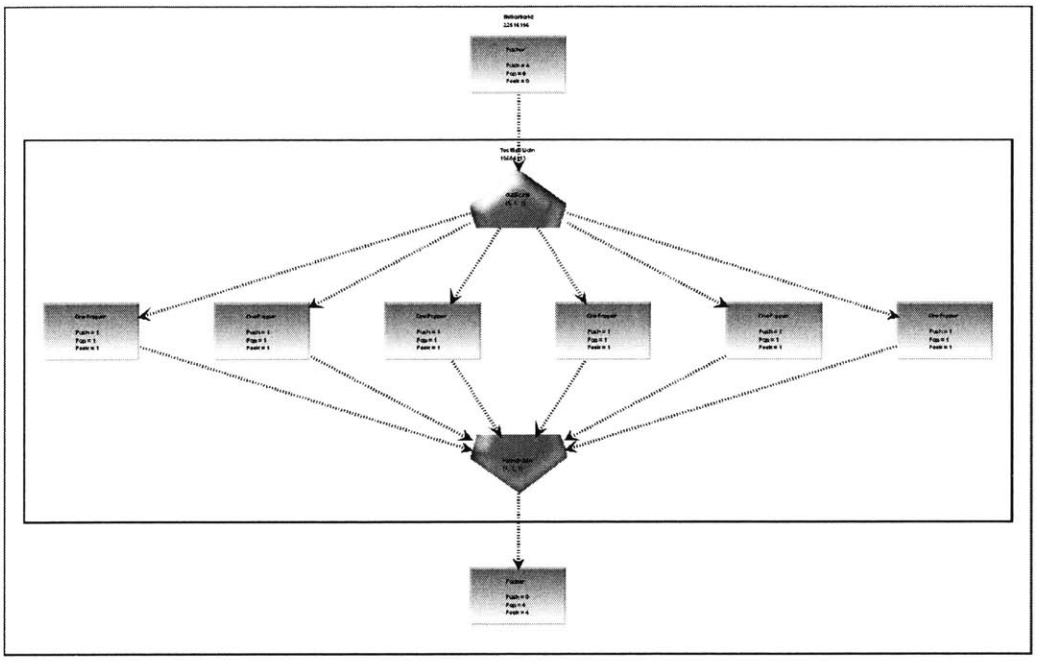


Figure 4-8: Graph corresponding to template code sample

code overwrites the existing file with the StreamIt source code. As a result, comments and other user changes in the source code are not preserved. The SGE would have to parse the entire source code file in order to detect the place where user modifications have occurred. A different underlying representation of the graph is needed to support immediate changes and move closer to the goal of complete isomorphism.

4.2.12 Exporting Graph Images

The SGE allows the creation of JPEG, GIF, PNG, image maps, and graphviz images of the graph. Thus, the developer can have the stream graphs created by the SGE available off-line. A physical copy of the graph can be obtained by using the SGE's print option. The print option is useful if the user does not want to create a graph image only to print it. No matter which option is used, the SGE's functionality allows users to have better control on how they can analyze their stream graphs.

4.3 Future Work

There are several things that still can be done in order to improve the current functionality of the prototype SGE [4]. One of the most extensive changes includes increasing the level of "isomorphism" between the StreamIt source code and its graphical representation. As it was mentioned before, this can be achieved by changing the internal representation of the code and the graph to something that can easily reflect changes (i.e. XML). Another change includes the duplication of large, complex components. The current implementation must be optimized so that there are no delays when any component is duplicated.

Another relevant change is to integrate the SGE completely into Eclipse. Currently, the SGE opens in a separate window from Eclipse. The SGE is not fully integrated yet because Eclipse does not support Swing/AWT. Eclipse has its own toolkit called SWT that deals with the graphical interface. Future versions of Eclipse might provide support for Swing/AWT and SWT interoperability.

In addition, now that the basic framework for the SGE has been created, more focus can be placed on exploring different features that could be useful to the StreamIt developer from input and comments received from users.

Chapter 5

Concluding Remarks

This thesis presents a graphical programming tool that can improve the performance of developers using the StreamIt programming language. The intuitive visualization of a StreamIt application permits the programmer to rapidly view the organization, flow of data and hierarchy in the stream graph. Various options in the SGE also allow the user to customize the view of the graph to whatever is more appropriate.

Automating the process of software development eliminates the possibility for human errors. The StreamIt Graph Editor can detect that the user is performing an illegal action and will not allow it. In other cases, the SGE simplifies tasks by automating all the work that the programmer would otherwise have to do manually.

The current version of the prototype contains many of the properties desired for rapid prototyping of stream programs with a focus on DSP applications. This thesis makes great strides towards more productive programming in the new and increasingly popular stream computing paradigm.

Appendix A

Code

This section contains the StreamIt code samples used in this thesis.

```
//Write the generated number to the filter's output tape.
void->int filter IntSource {
    int x;
    init {
        x = 0;
    }
    work push 1 {
        push(x++);
    }
}
// Print the number taken from the filter's input tape.
int->void filter IntPrinter {
    init {
    }
    work pop 1 {
        print(pop());
    }
}
// Write the 1st element in the filter's input tape to output tape.
int->int filter Original {
    init {
    }
    work push 1 pop 1 {
        push(pop());
    }
}
// Write 2nd element in the filter's input tape to output tape.
int->int filter Delayed {
    init {
    }
    work push 1 pop 2 {
        pop();
        push(pop());
    }
}
// Produce a delayed version of the data along with the original.
 int->int feedbackloop Echo() {
    join roundrobin(1, 1);
    body Original();
    loop Delayed();
    split duplicate;
    enqueue(0);
    enqueue(0);
}
// Add the first two elements in the filter's input tape.
int->int filter Adder {
    init {
    }
    work push 1 pop 2 {
        int a = pop();
        int b = pop();
        push(a + b);
    }
}

// Toplevel pipeline.
void->void pipeline EchoEffect2() {
    add IntSource();
    add Echo();
    add Adder();
    add IntPrinter();
}
```

Figure A-1: Source code for the EchoEffect StreamIt program.

Appendix B

Figures

This section contains the graphs belonging to the StreamIt source programs that were shown as examples in this document.

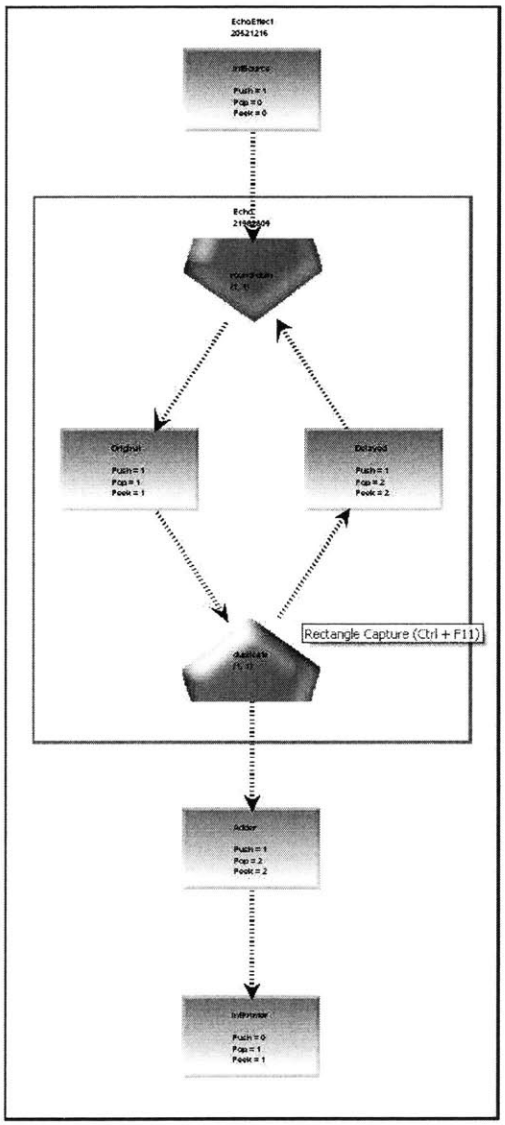


Figure B-1: Graph representation of the EchoEffect program

Bibliography

[1] Gaudenz Alder. Design and Implementation of the JGraph Swing Component. Technical report, JGraph, 2003.

[2] Scott Fairbrother. Using and Extending Eclipse. Workshop at the Massachusetts Institute of Technology, September 2003.

[3] IBM Eclipse Group. Eclipse Platform Technical Overview. Technical report, IBM, 2003.

[4] StreamIt Group. StreamIt webpage. http://catfish.csail.mit.edu/streamit/.

[5] National Instruments. *LabVIEW User Manual.* National Instruments Corporation, Silicon Valley, 7.0 edition, April 2003.

[6] Mikls Marti kos Ldeczi and Pter Vlgyesi. The Generic Modeling Environment Technical Report. Technical report, Vanderbilt University, 2003.

[7] The MathWorks. Simulink. Technical report, The MathWorks, Inc., 2003.

[8] The MathWorks. Texas Instruments Streamlines Research and Development with Simulink and DSP Tools. *MathWorks User Story*, 2004.

[9] Sherry Shavor, Jim D'Anjou, Scott Fairbrother, Dan Kehn, John Kellerman, and Pat McCarthy. *The Java Developers Guide to Eclipse.* Addison-Wesley, 2003.

[10] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A Language for Streaming Applications. In *International Conference on Compiler Construction*, Grenoble, France, April 2002.

[11] William Thies, Michal Karczmarek, Michael Gordon, David Z. Maze, Jeremy Wong, Henry Hoffman, Matthew Brown, and Saman Amarasinghe. StreamIt: A Compiler for Streaming Applications. MIT/LCS Technical Memo MIT/LCS Technical Memo LCS-TM-622, Massachusetts Institute of Technology, Cambridge, MA, December 2001.

[12] Claire Tristram. Everyone's a Programmer. *Technology Review*, 2003.