The StreamIt Development Tool: A Programming Environment for StreamIt

by

Kimberly S. Kuo

Submitted to the
Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Electrical Engineering and Computer Science

and

Master of Engineering in Elecrical Engineering and Computer Science

at the Massachusetts Institute of Technology

May 20, 2004

Author .....
Department of Electrical Engineering and Computer Science
May 20, 2004

Certified by .....
Saman Amarasinghe
Associate Professor
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

The StreamIt Development Tool:

A Programming Environment for StreamIt

by

Kimberly S. Kuo

Submitted to the
Department of Electrical Engineering and Computer Science

May 20, 2004

In Partial Fulfillment of the Requirements for the Degrees of
Bachelor of Science in Electrical Engineering and Computer Science
and
Master of Engineering in Elecrical Engineering and Computer Science

ABSTRACT

StreamIt [28] is a high-level programming language intended for the development of large-scale and high-performance streaming applications that are characterized by the processing of data streams by modular structures. The StreamIt Development Tool (SDT) [25] is designed to aid the coding and simultaneous code- and graph-based debugging and visualizing of programs written in StreamIt. The goal is to provide a graphical programming environment that simply and intuitively conveys the hierarchical and structured nature of the StreamIt language by visually interpreting the dynamic behavior and graph representation of a StreamIt application. Consequently, the SDT provides utilities for program creation and code editing, compilation and launch support, breakpoints and code stepping, general debugging infrastructure, help support, stream graph examination and navigation, and stream data display, modification, and tracking. A user study evaluating the SDT uncovered several problems and areas of improvement that need to be addressed before this tool can approach its goals. Assessment of the SDT's efficacy in its current state is inconclusive—the SDT demonstrates both the ability to improve and hinder a user's debugging ability. Facilitating effective coding and debugging techniques and developing for scalability are critical elements in improving the SDT's effectiveness.

Thesis Supervisor: Saman Amarasinghe
Title: Associate Professor

Acknowledgments

I would like to thank my thesis advisor, Saman Amarasinghe, for his guidance and support throughout the development of this thesis. I am also thankful to the StreamIt group at MIT, from whom I have learned so much about StreamIt and Eclipse: Rodric Rabbah coordinated and assisted in the design of the user study and provided invaluable feedback for the SDT's functionality and this thesis. William Thies suggested and/or wrote many of the applications in the user study and supplied advice on improving and enhancing the SDT and the user study. Both David Maze and Bill Thies modified the StreamIt Java library to interface with the SDT. Jasper Lin, Juan Carlos, Sitij Agarwal, Jeremy Wong, Michael Gordon, and Michal Karczmarek all participated in in-house evaluations of the SDT.

Finally, I would like to thank my parents, Kuonan and Sue, and my brother, Raymond. I am eternally grateful for their support and encouragement in both my academic and personal endeavors. I could not have made through MIT or this thesis without them.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Overview

Streaming applications encompass a set of programs characterized by large data streams, independent modular transforms called filters that can execute on these streams concurrently, a predictable pattern of applying certain transforms, some dynamic change in the pattern of transform application, some inter-filter communication, and high-performance [40]. Streaming applications are prevalent in desktop applications such as streaming media, software radio, real-time encryption, and graphics packages; hand-held computers, cell phones, and DSPs in the embedded domain; and high-performance systems such as internet routers, cell phone base stations, and multimedia editing consoles [29] [30] [31] [40].

StreamIt [28] is a high-level programming language intended for large-scale and high-performance streaming application development. This language introduces abstractions, such as hierarchically structured streams that are designed to facilitate modularity, programmer productivity, and robustness [29]. StreamIt aims to allow programmers to easily and naturally express their design of streaming applications. The StreamIt compiler generates a representation of a streaming program, called a stream graph, which is a directed graph of filters connected by streams. A streaming application developer can analyze these graphs separately from program execution [27].

In partnership with creation and design, the debugging, incremental building, and performance optimization of large streaming applications necessitates utilities for stream graph visualization, data tracking, and filter execution order. The StreamIt Development Tool (SDT) [25] is designed to aid the coding and simultaneous code- and graph-based debugging and visualizing of programs written in StreamIt. The SDT's goal is to provide a graphical programming environment that simply and intuitively conveys the hierarchical and structured nature of the StreamIt language by interpreting and visually representing the stream graph and the dynamic behavior of a StreamIt program. To address this goal, the SDT needs to provide common code editing and debugging functionality analogous to the functionality available in traditional application development environments. As such, the SDT supplies the following features:

- A StreamIt file editor with customizable syntax highlighting
- Utilities for creating and managing StreamIt applications
- Line breakpoint, method breakpoint, and watchpoint addition to executable code
- Breakpoint property specification, such as enabling, disabling, conditional breaking, etc.
- Program launching and compilation support with syntax error detection and reporting
- Program suspension and code stepping
- Variable inspection and value modification
- Help manual with tutorial

Moreover, the SDT offers features tailored to the StreamIt language's syntax and semantics and targeted at debugging the salient quality of streaming applications—the ability to easily represent a streaming program graphically, structurally, and hierarchically:

- Stream graph examination and hierarchical navigation
- Stream statistical information display
- Data inspection and modification
- Dataflow tracking

Built in a visual, integrated development environment (IDE) with a human-computer interface, the SDT is composed of the following modules: an IDE-integrated debugger, graphical text editor, a runtime stream graph view, and a corresponding graph overview. The SDT is implemented in Java as an Eclipse [23] plug-in.

1.2 Organization

The rest of this thesis is organized as follows. Related work is presented in chapter 2: debugging and visualizing tools for high-level (2.1) and object-oriented programming (OOP) languages (2.2). In chapter 3, appropriate background material on StreamIt (3.1) and Eclipse (3.2) is provided. Chapter 4 describes common coding and debugging features of the SDT: program creation and code editing (4.1), breakpoints (4.2), program compilation and launch support (4.3), and general debugging (4.4) and help (4.5) support. In chapter 5, the SDT's StreamIt- and stream graph-specific features are given: stream graph examination and navigation (5.1) and stream data display, modification, and tracking (5.2). Chapter 6 presents a user study of the SDT: analysis of the target user population (6.1), procedures (6.2), an in-house evaluation (6.3), what occurred during the study (6.4), quantitative results (6.5), qualitative results (6.6), and lessons learned (6.7). Future research opportunities are discussed (7.1) and this thesis is concluded with (7.2).

Chapter 2

Related Work

In this chapter, related work is presented on tools for debugging and visualizing high-level (2.1) and object-oriented programming (OOP) languages (2.2).

2.1 High-Level Language Debuggers and Program Visualization Tools

Although the theory of streaming applications dates back to the 1960s [34], the majority of streaming applications are traditionally written in various assembly languages [30]. However, as these applications increase in size, functionality, and complexity, high-level languages such as C are being used to increase programmer productivity, improve program reliability and readability, and decrease development and debugging time [30]. In conjunction, technical computing tools like Matlab [2] are often employed for algorithm development and verification [18].

Numerous debuggers and program visualization tools exist for streaming applications written in high-level languages. The majority of these tools are C debuggers targeted at specific hardware or platforms, offering traditional debugging features (i.e. program suspension, breakpoint stepping, watchpoints, local variable and output display, etc.) combined with assembly code, memory register, and signal plot display. Indeed, screenshots of those described below reveal this similarity, as seen in

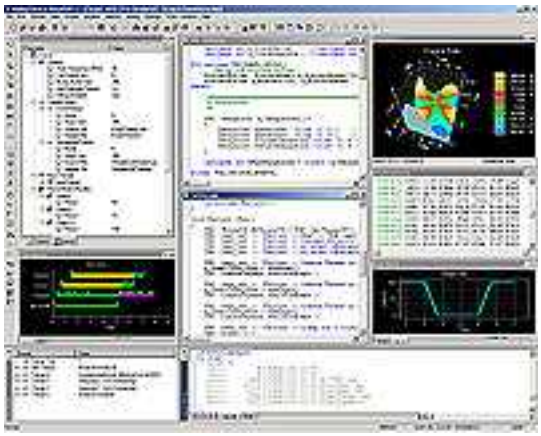


Figure 2-1: Screenshot of the VisualDSP++™ [42].

Figures 2-1 through 2-6.

VisualDSP++™ [42] is an IDE and debugger developed by Analog Devices, Inc. [10] for building and debugging DSP applications written in C/C++ and assembly (Figure 2-1). It produces executables that can run on simulators and emulators. IAR Systems' [35] Embedded Workbench [44] is another IDE intended for C/C++ embedded applications (Figure 2-2). Created by Cradle Technologies, Inc. [38], Rapid Development System (RDS) [7] is a software development kit (SDK) for developing and debugging DSP applications in C (Figure 2-3). After compilation, the RDS can debug using simulators or proprietary hardware. BoxView [6] and Debug-56K [9] are two debuggers built by Domain Technologies, Inc. [39] for DSP-targeted C and assembly applications (Figure 2-4). Both typically interface with emulators, which in turn interact with third-party DSPs. QNX Software Systems' [36] Momentics® [19] is an Eclipse-integrated IDE for real-time operating systems (RTOS) and embedded applications in C, C++, and Java (Figure 2-5). Unlike RDS, Momentics' source code debugger functions separately from target hardware interaction. Finally, the Debugger RTOS [4] is a RTOS built-in module aimed at embedded software (Figure 2-6).

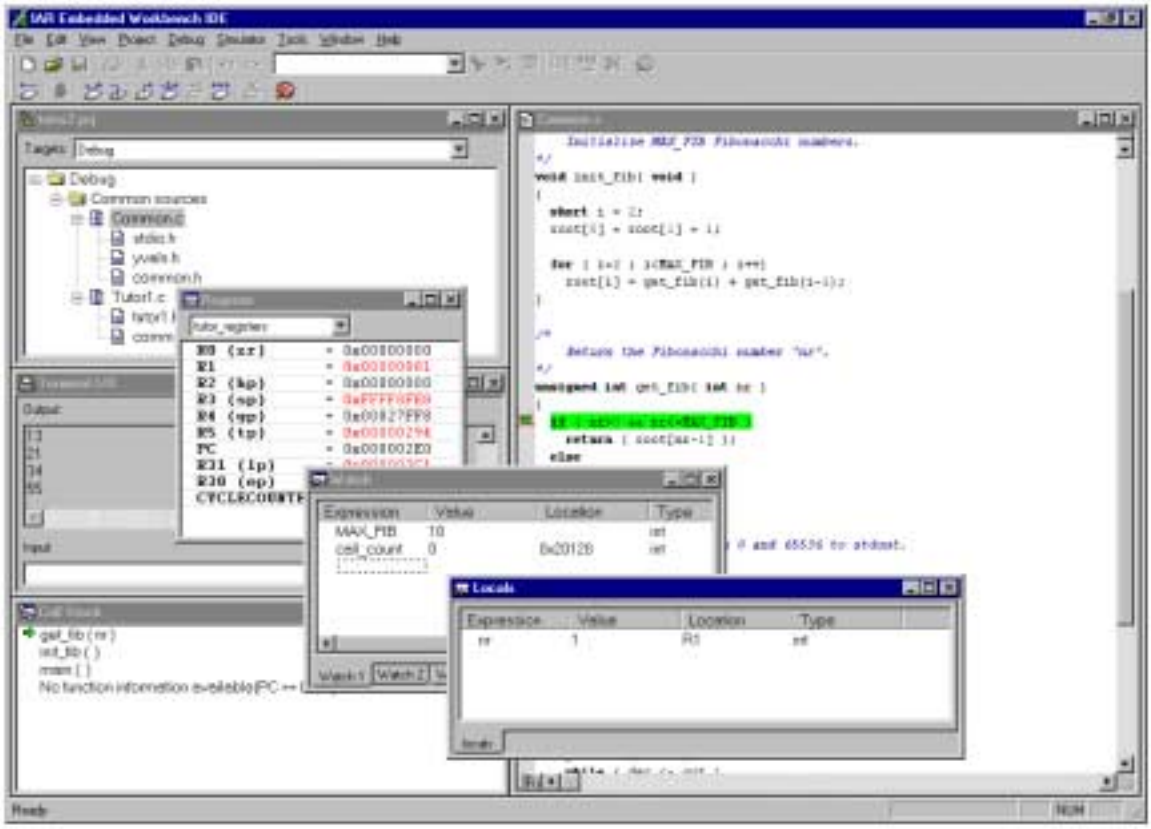


Figure 2-2: Screenshot of the Embedded Workbench [32].

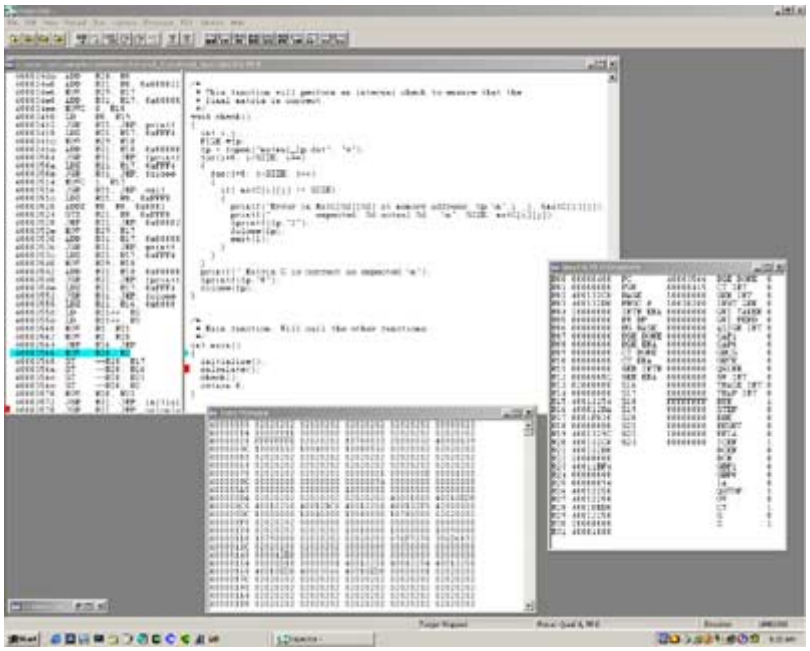


Figure 2-3: Screenshot of the Rapid Development System [37].

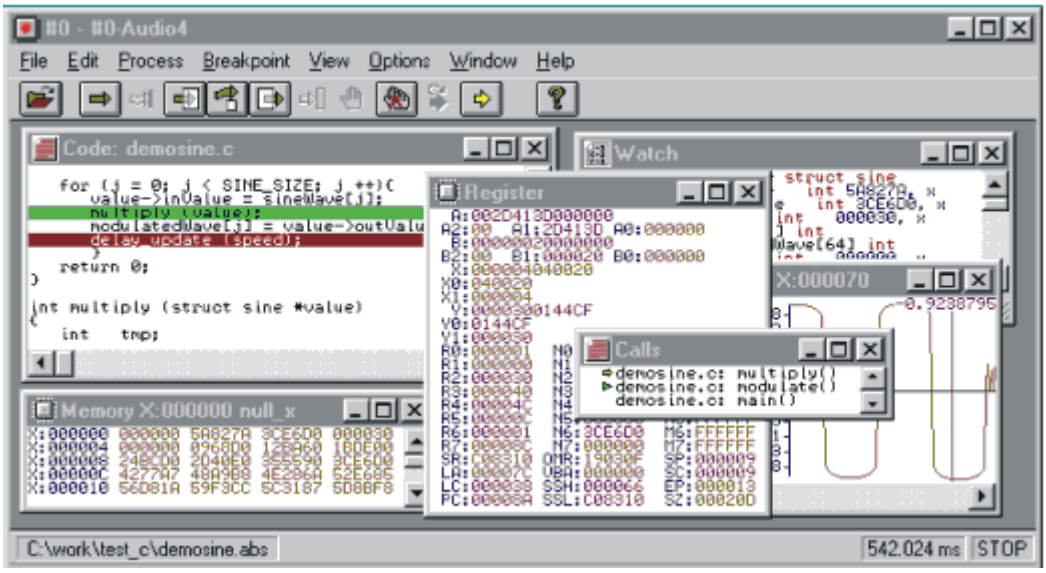


Figure 2-4: Screenshot of the BoxView [6].

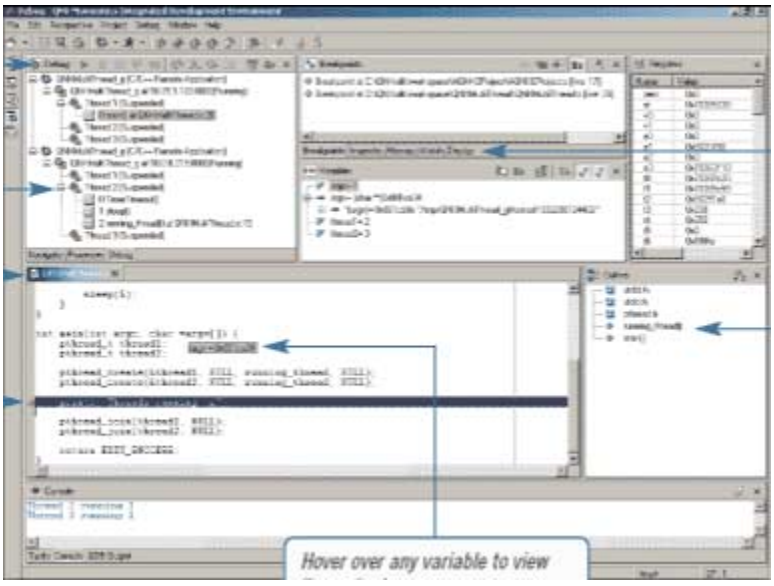


Figure 2-5: Screenshot of the Momentics® [33].

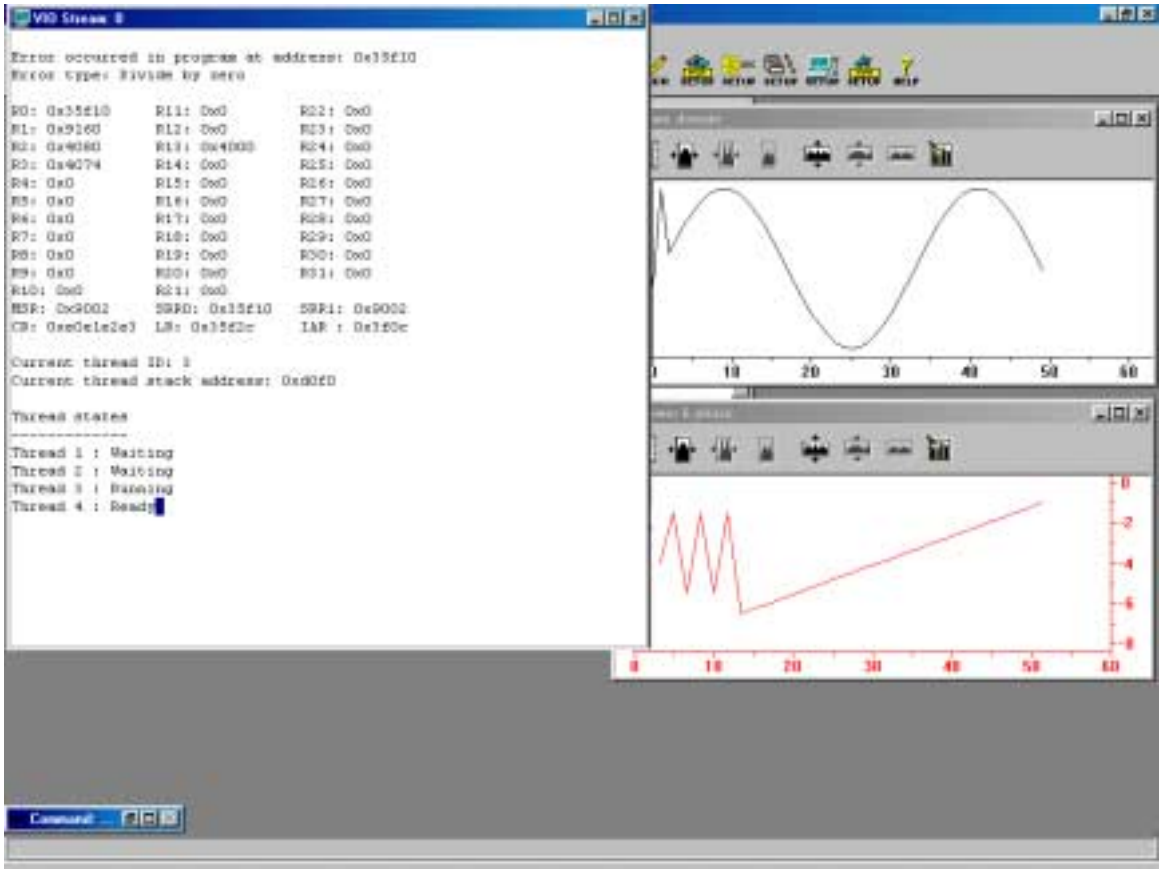


Figure 2-6: Screenshot of the Debugger RTOS [4].

2.2 OOP Debuggers and Program Visualization Tools

In recent years, some movement in the streaming domain has been made towards OOP languages such as C++ or Java, which introduce abstractions that improve the portability and reusability of code [30]. The introduction of conceptual abstractions empowers the design, debugging, visualization, and analysis tools created for OOP-based streaming applications to introduce hierarchical, modular structures while hiding unnecessary details from the programmer. On top of the traditional debugging features previously mentioned, all three of the tools described next use some variation on the theme of signal processing blocks that are connected, displayed, and navigated graphically.

Simulink [1] is a modeling, simulation, and analysis tool for control, signal processing, and communications system design. This tool imposes OOP conventions on Matlab, C, Fortran, and Ada programmers by allowing its users to insert their code into the methods of pre-defined blocks or to use application-specific standard block libraries. Furthermore, hierarchically block navigation at both the design and debugging stages is offered: command-line Simulink Debugger enables breakpoint stepping of the currently executing method which is simultaneously displayed on its associated block. Additional information, such as block state, inputs, and outputs, are visible in other windows.

Process-Level Debugger (PDG) [8] is designed for a graphical parallel programming environment for concurrent applications called GRAPE. The PDG models processes as black boxes that interact with each other, as illustrated in Figure 2-7. Like Simulink, programmers build their applications by creating and connecting black boxes hierarchically (i.e., each black box may be composed of sub-boxes—sub-processes—and displayed in a graphical view). As an application is debugged, the PDG shows the application's behavior on this view and allows a programmer to zoom down on suspicious process blocks in the hierarchy. This top-down debugging method can eventually find the associated erroneous code, as seen in Figure 2-8.

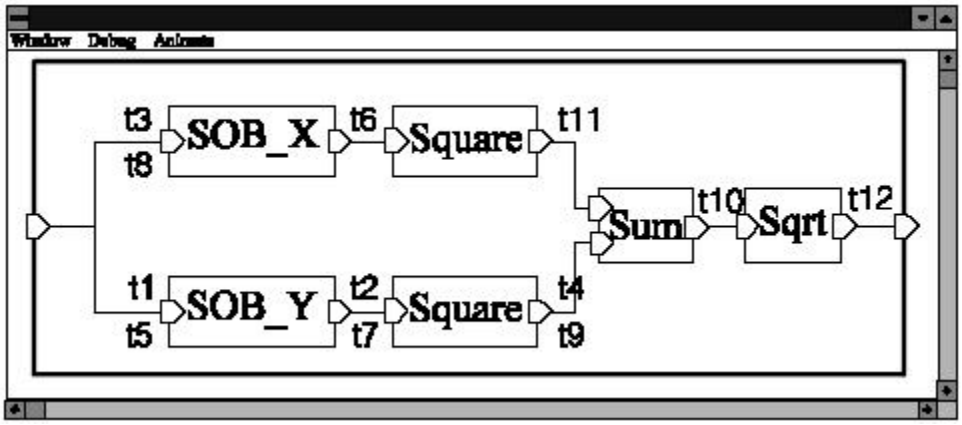


Figure 2-7: Graphical view of an application created with the PDG [8].

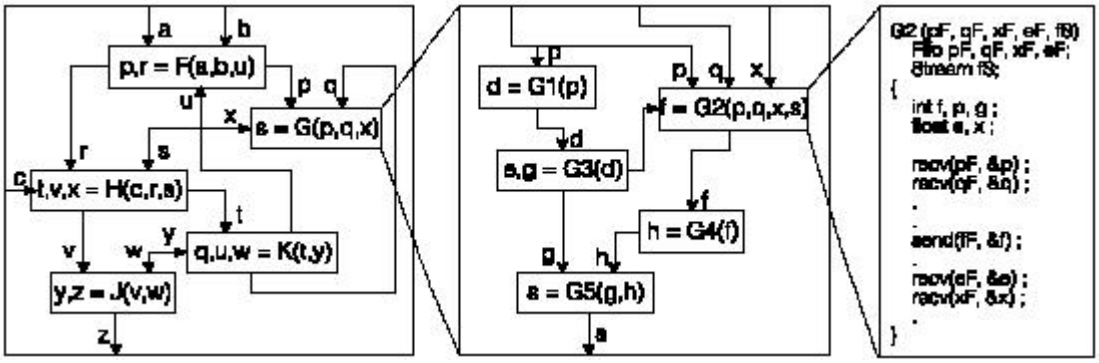


Figure 2-8: The PDG's hierarchical graphical representation of an application showing successively lower levels of the hierarchy from left to right [8].

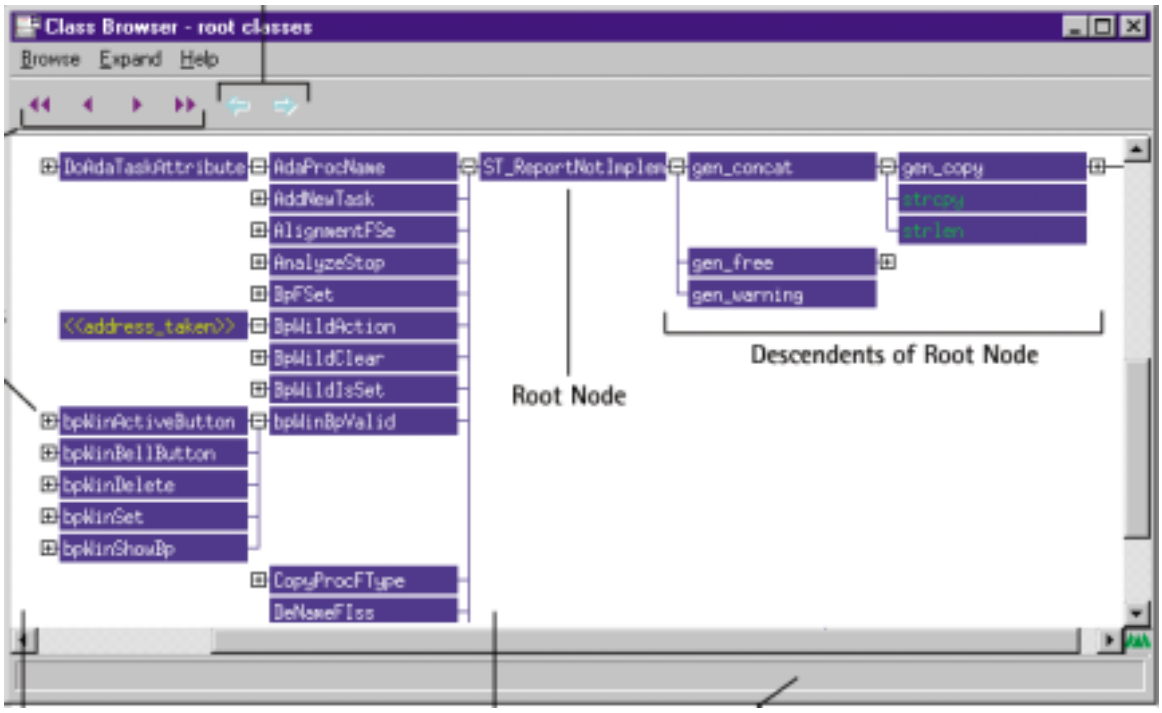


Figure 2-9: Screenshot of the MULTI® Integrated Development Environment [20].

The MULTI® Integrated Development Environment [12] produced by Green Hills Software, Inc. is designed for multiprocessor, distributed systems and embedded applications using C, C++, Ada, Fortran, and assembly. Besides standard editing and debugging functionality, this IDE conveys program control flow with perusable static and dynamic call graphs and class hierarchies, as seen in Figure 2-9.

Chapter 3

Background

Unlike the programming languages described in the previous chapter, StreamIt is specifically designed for streaming applications. This chapter formally introduces the StreamIt language (3.1). This thesis contributes the first IDE for StreamIt and does so as a plug-in to Eclipse, a universal tool platform from IBM. Relevant background on the infrastructure and functionality of Eclipse is given (3.2).

3.1 The StreamIt Language

Since the language-specific features of StreamIt determine much of the SDT's features and functionality, this section provides an overview of the syntax and semantics of StreamIt version 2.0 [26, 40]. StreamIt programs are composed of computational units that process large streams of data. These units, called *filters*, can be hierarchically composed into *pipelines*, *split-joins*, and *feedback-loops*. A *stream* is defined to be a filter, pipeline, split-join or feedback-loop. *Channels* provide the means by which data is communicated between streams.

3.1.1 Channels

Channels are one-way FIFO queues that pass data of a specific type from one stream to another. Each stream has one input channel from which it can read data, and one

output channel to which it can write data. A channel's *source* stream writes data to it, while a channel's *sink* stream reads data from it. The data type of a channel can be one of the following: boolean, int, float, one-bit integer, complex number. Furthermore, a channel's type can also be fixed-length arrays of any of the data types previously mentioned.

3.1.2 Filters

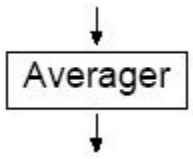
The fundamental computational unit in StreamIt is the filter, depicted in Figure 3-1. A filter's behavior is defined by its `init`, `prework`, and `work` functions, which contain C-style code. The `init` function executes only once at the start of a program, the `prework` runs in place of the `work` function the first time the work function should execute, and the work function runs repeatedly thereafter. The code within each function can call helper functions and access and modify state variables, which are the non-local variables of stream. However, only the `prework` and `work` functions can call the three special functions that read from or write to the I/O channels: `push(x)` pushes a datum with value `x` onto the output channel's queue, `pop()` pops and returns the last datum from the input channel's queue, and `peek(n)` returns the value of the datum at the `n`-th index on the input channel's queue. For example, `peek(0)` returns the value of the next datum that could be popped. The `prework` and `work` functions must declare their push, pop, and peek[1] rates—the number of data they push, pop, and peek during each execution.

3.1.3 Pipelines

A pipeline, seen in Figure 3-2, is a simple container structure composed of one or more child streams connected sequentially. Within a pipeline, each stream's output channel is chained to the stream's input channel. The first child's input channel is linked to the pipeline's input channel, while the last child's output channel is linked to the pipeline's output channel. Children are specified with `add` statements. Although a pipeline does

[1] A `prework` and `work` function's peek rate is the maximum index at which the function peeks

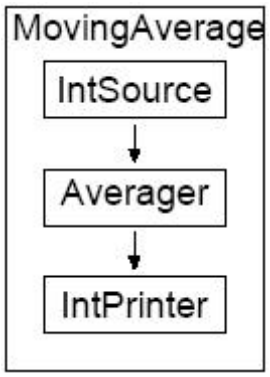
```
int->int filter Averager(int n) {
    work pop 1 push 1 peek n {
        int sum = 0;
        for (int i = 0; i < n; i++)
            sum += peek(i);
        push(sum/n);
        pop();
    }
}
```



(a) (b)

Figure 3-1: The code (a) and conceptual representation (b) of an `Averager` filter.

```
void->void pipeline MovingAverage {
    add IntSource();
    add Averager(10);
    add IntPrinter();
}
```



(a) (b)

Figure 3-2: The code (a) and conceptual representation (b) of a `MovingAverage` pipeline with three children.

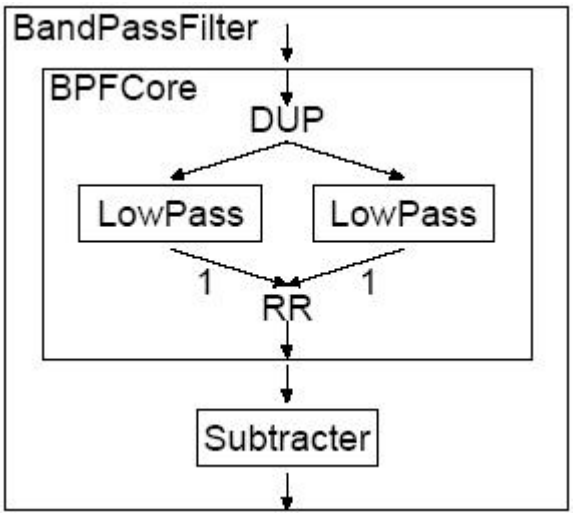
not have formal `init`, `prework`, or `work` functions, and its declaration effectively acts as an `init` function, which can contain code that parameterizes its composition. The execution of a pipeline (like that of a split-join or feedback-loop described next) can be thought of as the composition of all its children's work functions.

3.1.4 Split-joins

A split-join, depicted in Figure 3-3, models parallel computation by linking its input channel to a *splitter*, which redistributes data to input channels of the child streams, and linking its output channel to a *joiner*, which recombines data from the children's output channels. A split-join has two or more children. Splitters are created with

```
float->float pipeline BandPassFilter(
    float rate, float low,
    float high, int taps) {
    add BPFCore(rate, low,
                high, taps);
    add Subtracter();
}
float->float splitjoin BPFCore(
    float rate, float low,
    float high, int taps) {
    split duplicate;
    add LowPass(rate, low, taps, 0);
    add LowPass(rate, high, taps, 0);
    join roundrobin;
}
float->float filter Subtracter {
    work pop 2 push 1 {
        push(peek(1) - peek(0));
        pop(); pop();
    }
}
```

(a)



(b)

Figure 3-3: The code (a) and conceptual representation (b) of a `BPFCore` splitjoin with two children.

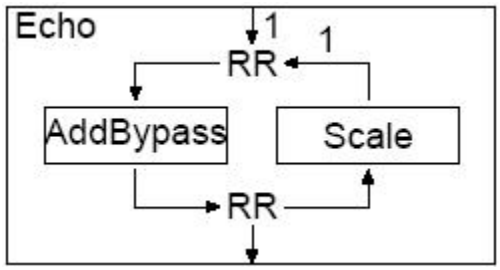
`split` statements and are either `duplicate` or `roundrobin(`x_1`, `x_2`, ..., `x_n`)`—the former distributes a copy of each datum to each child, while the latter distributes x_1 data to the first child, x_2 data to the second child, etc. Joiners are created with `join` statements and are only of type `roundrobin(`x_1`, `x_2`, ..., `x_n`)`—takes x_1 data from the first child, x_2 data from the second child, etc. Children are specified with `add` statements, with the `i`-th `add` statement corresponding to a split-join's `i`-th child. As in Figure 3-3, the children are usually drawn left to right, first to last. Furthermore, like a pipeline, a split-join does not have formal `init`, `prework`, or `work` functions, but its declaration in effect is an `init` function.

3.1.5 Feedback-loops

In order to models cycle, a feedback-loop, seen in Figure 3-4, is composed of a joiner, a splitter, and two child streams. The first child acts as the body of the cycle while the second child acts as the loop. The joiner combines data from the loop's output channel and the feedback-loop's input channel and sends it to the body's input channel.

```
float->float feedbackloop Echo(int n,
    float f) {
    join roundrobin(1, 1);
    body FloatAdderBypass();
    loop float->float filter {
        work pop 1 push 1 {
            push(pop() *  f);
        }
    };
    split roundrobin;
    for (int i = 0; i < n; i++)
        enqueue(0);
}
float->float filter FloatAdderBypass{
    work pop 2 push 2 {
        push(peek(0) + peek(1));
        push(peek(0));
        pop(); pop();
    }
}
```

(a)



(b)

Figure 3-4: The code (a) and conceptual representation (b) of an `Echo` feedbackloop.

The splitter, which is linked to the body's output channel, distributes data to the loop's input channel and the feedback-loop's output channel. Splitters and joiners are created in the same way and have the same types as in split-joins. The first child is specified with a `body` statement, the second with a `loop` statement. As in Figure 3-4, the children are usually drawn left to right, first to last. Furthermore, like a pipeline, a feedback-loop does not have formal `init`, `prework`, or `work` functions, but its declaration is really an `init` function. This code is also the place where `enqueue` statements can be used to push data onto the loop's output channel at the start of a program.

3.1.6 Hierarchical Graph Representation

As seen in Figures 3-1 through 3-4, a StreamIt program can be visually depicted as a hierarchical directed graph of streams, with graph nodes representing streams and graph edges representing channels. The unique root of a stream graph is defined to be a top-level pipeline—a pipeline with no input or output channels. Because the StreamIt language allows its programmers the ability to define a stream element once

but use it many times throughout the code, a one-to-many correspondence exists between the code and graph representations of a StreamIt program. Accordingly, the code of a stream shall be called a *declaration*, while the graph node of a stream shall be called an *instance*.

3.1.7 Execution Model

When a StreamIt application is initially executed, the `init` function of each stream is executed in a depth first search order starting at the top-level pipeline of the hierarchical stream graph. Because the `init` functions contain the code to specify child streams, these functions build the stream graph when executed. Once all the `init` functions are finished, all channels are created and connected. Afterwards, the `prework` and `work` functions are executed until program completion.

In order to generate an executable program, the StreamIt compiler is responsible for deriving a schedule of filter execution—the order in which the work function of each filter in the stream graph can be executed ("fired") so that each filter's peek, pop, and push rates are satisfied. The number of times that a filter fires during program execution is also dependent on the programmer-specified number of iterations of the top-level pipeline.

The StreamIt compiler also generates a file containing a static stream graph that programmers can analyze separate from its execution (Figure 3-5).

3.2 Eclipse

The Eclipse Platform [24] was chosen as the SDT's implementation toolkit because it is designed for building integrated development environments (IDEs) that can be used to create a diverse array of applications on Eclipse. As such, the SDT makes heavy use of a user interface (UI) for editing files and viewing file and program metrics, a resource management model and help system, GUI and UI libraries, and language-independent launching, running, and debugging infrastructure provided by Eclipse.

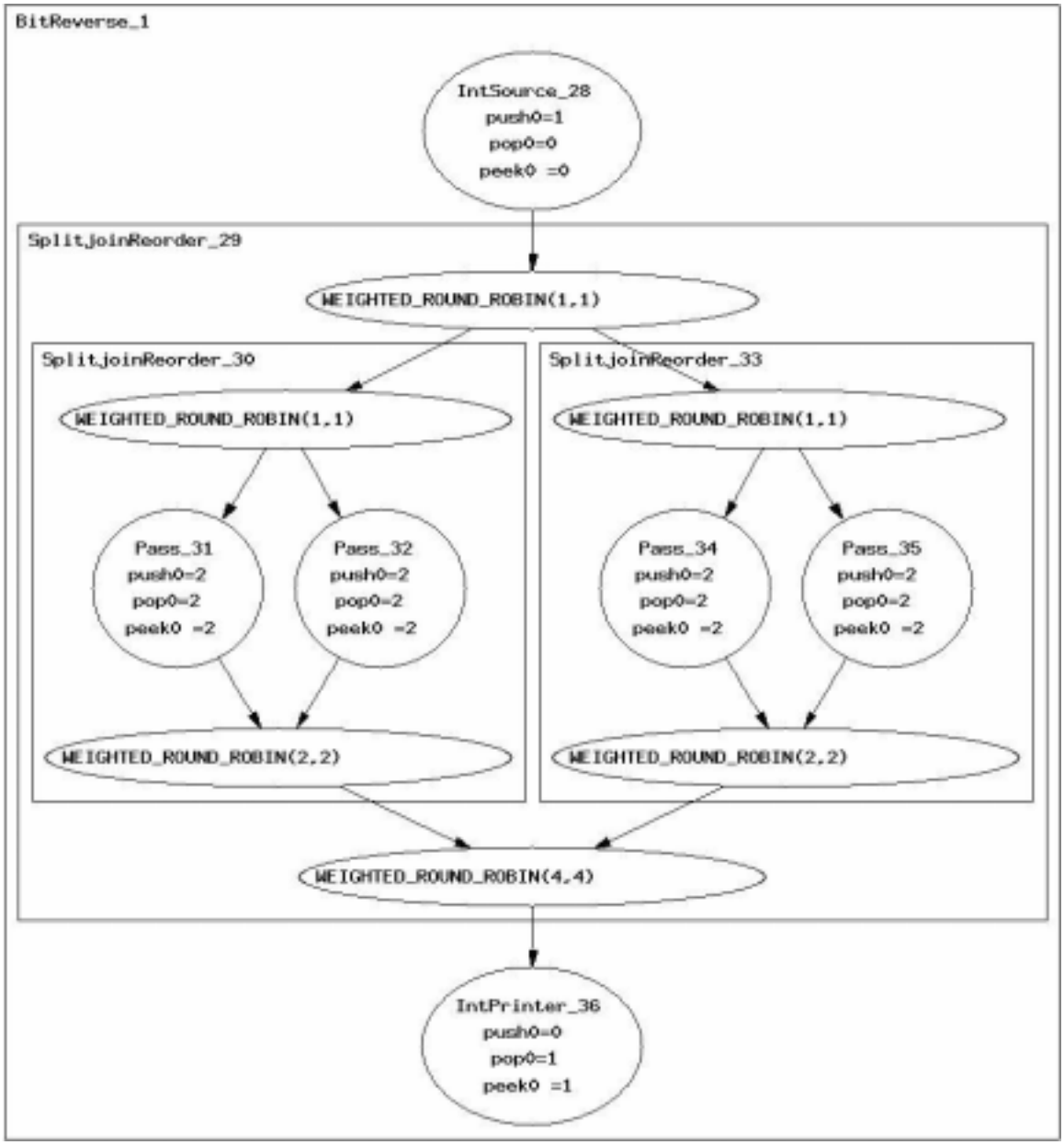


Figure 3-5: A static stream graph generated by the StreamIt compiler.

Furthermore, Eclipse's ability to run on many platforms[2] was also considered because of the desire to make the SDT widely deployable. Accordingly, because much of the Eclipse-provided infrastructure and functionality influences the SDT's features, design, and implementation, this section provides a brief overview of Eclipse version 2.1.1 [22].

3.2.1 Plug-ins

The SDT is implemented as a plug-in for the Eclipse Platform. A *plug-in* is a module encapsulating the IDE (in this case, the SDT) to be added to the existing Eclipse Platform. Plug-ins operate on the IDE-specific files and add IDE-specific UI to the existing Eclipse UI. Plug-in usability and quality is contingent upon successful integration with the Eclipse Platform and upon how well different features of the plug-in work with each other [16].

3.2.2 Workbench

Depicted in Figure 3-6, Eclipse's general desktop environment is called the *Workbench*. Each Workbench window contains one or more perspectives, which in turn contain editors and views. A *perspective* defines the menus, toolbars, and views that appear within a Workbench window at a given time. Their purpose is to logically group menus, toolbars, and views related to specific user tasks. For instance, the Resource perspective seen in Figure 3-6 is intended for resource management, while the Debug perspective combines views related to debugging programs.

Perspectives generally dedicate an area to editors. Eclipse provides a primitive text editor seen in Figure 3-6. The Workbench window also contains a main menu bar and toolbar which contain items relevant to the current perspective and currently opened file. The Workbench provides a Preferences dialog for setting user preferences and a help system for browsing and searching documentation.

[2]Windows 98/ME/2000/XP, Red Hat Linux Version 7.1 (x86/Motif and x86/GTK), SuSE Linux 7.1 (x86/Motif and x86/GTK), Solaris 8 (SPARC/Motif), QNX (x86/Photon), AIX (PPC/Motif), HP-UX (HP9000/Motif), and Mac OSX (Mac/Carbon)

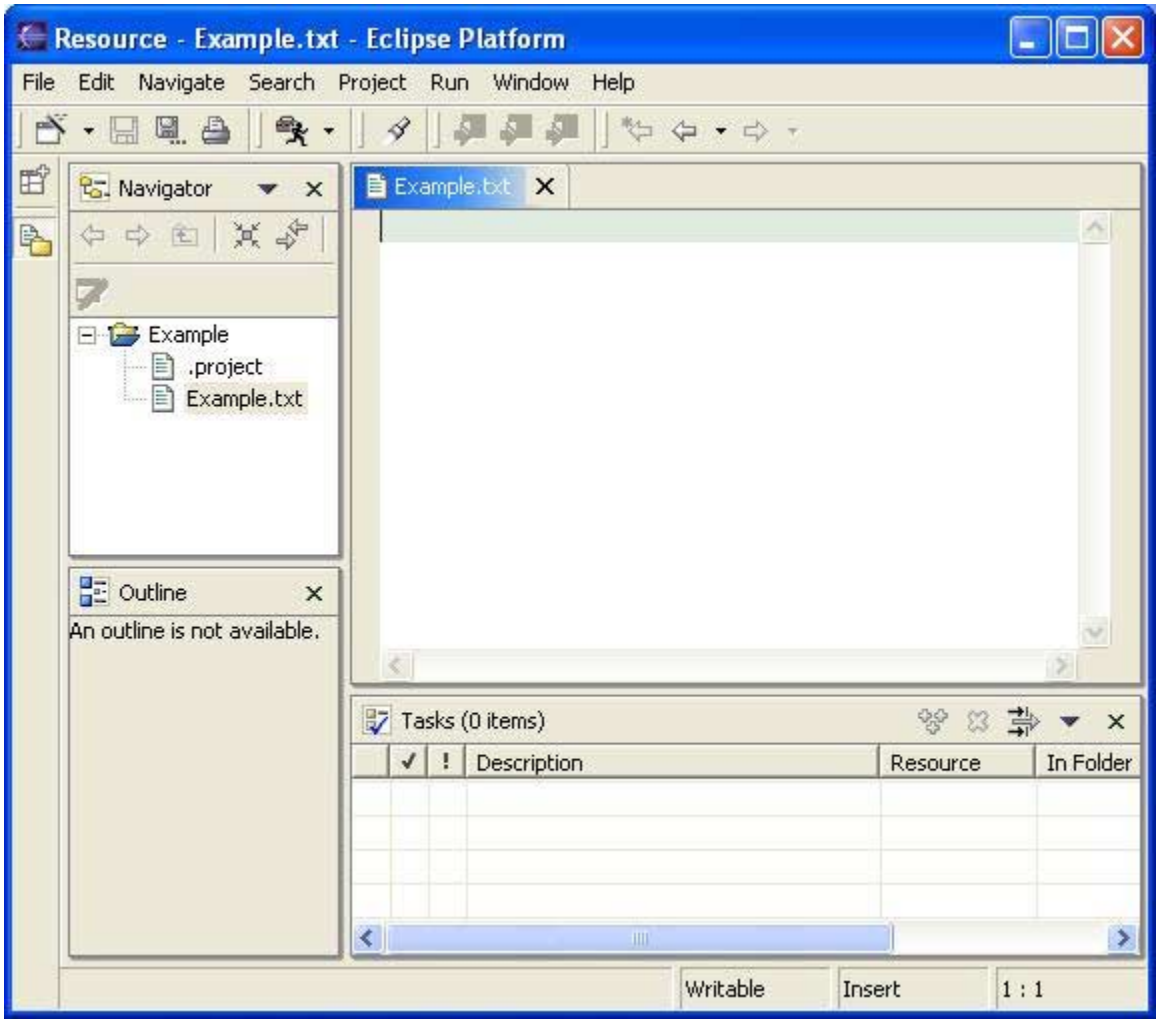


Figure 3-6: An Eclipse Workbench window in the Resource perspective, containing a text editor and the Navigator, Outline, and Tasks views.

Views display alternative presentations of files opened in editors and provide ways of navigating the information in your Workbench. Unlike editors, they have their own menus and toolbars. In Figure 3-6, there are three views of interest: The Navigator, Outline, and Tasks views. The Navigator view displays a hierarchical representation of the resources within the Workbench and allows users to perform various operations on files (e.g. add, remove, open, etc). Eclipse uses a resource management system consisting of projects, which contain files and folders. The Outline view displays an editor-specific outline of a file that is currently open. The Tasks view displays problems, errors, or warning related to Workbench resources.

3.2.3 Launching, Running, and Debugging Infrastructure

Eclipse enables users to manage how programs are launched for running or debugging with the Launch Configurations dialog depicted in Figure 3-7. Launch configurations can be created to configure how a program is launched, such as its arguments or the particular file within a project to be used.

A Debug perspective, depicted in Figure 3-8, is provided for users to run and debug their programs. There are three views of interest in Figure 3-8: the Debug, Variables, Console, and Properties views. The Debug view displays the processes, threads, and stack frames of the programs being run or debugged. The Variables view displays information, such as variable type or value, about variables in the stack frame currently being selected in the Debug view. The Console view displays the standard output, error, and input of processes that have been run or debugged. The Properties view displays properties of the view or editor that has focus (i.e., is selected) in the Workbench. Properties are displayed in a tabular format with one column for property names and another column for property values. The toolbar of the Properties view allows users to decide the sorting order of the properties, such as by category, with filters, default order, etc.

Finally, Eclipse comes pre-packed with Java Development Tooling (JDT) [41]. The JDT is a set of plug-ins that create an extensive IDE for Java application development.

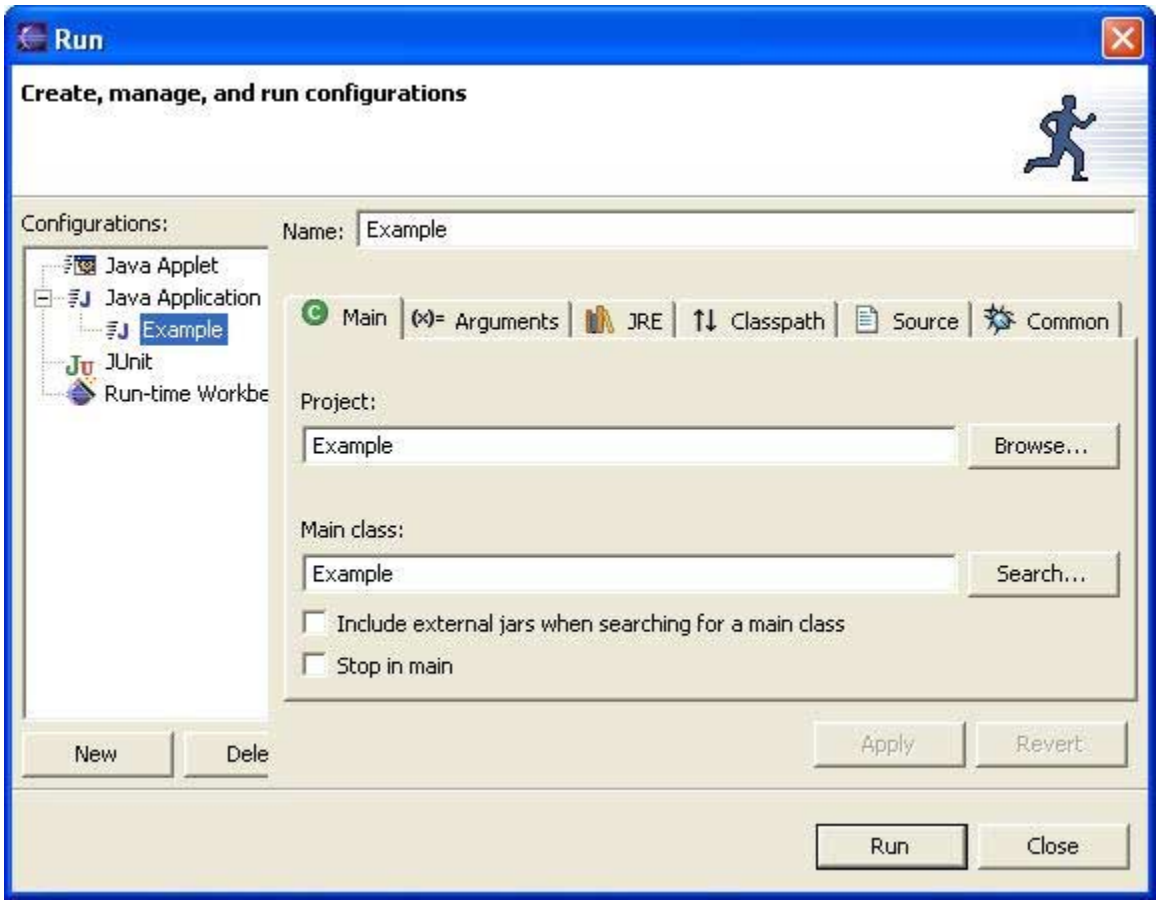


Figure 3-7: A Launch Configurations dialog.

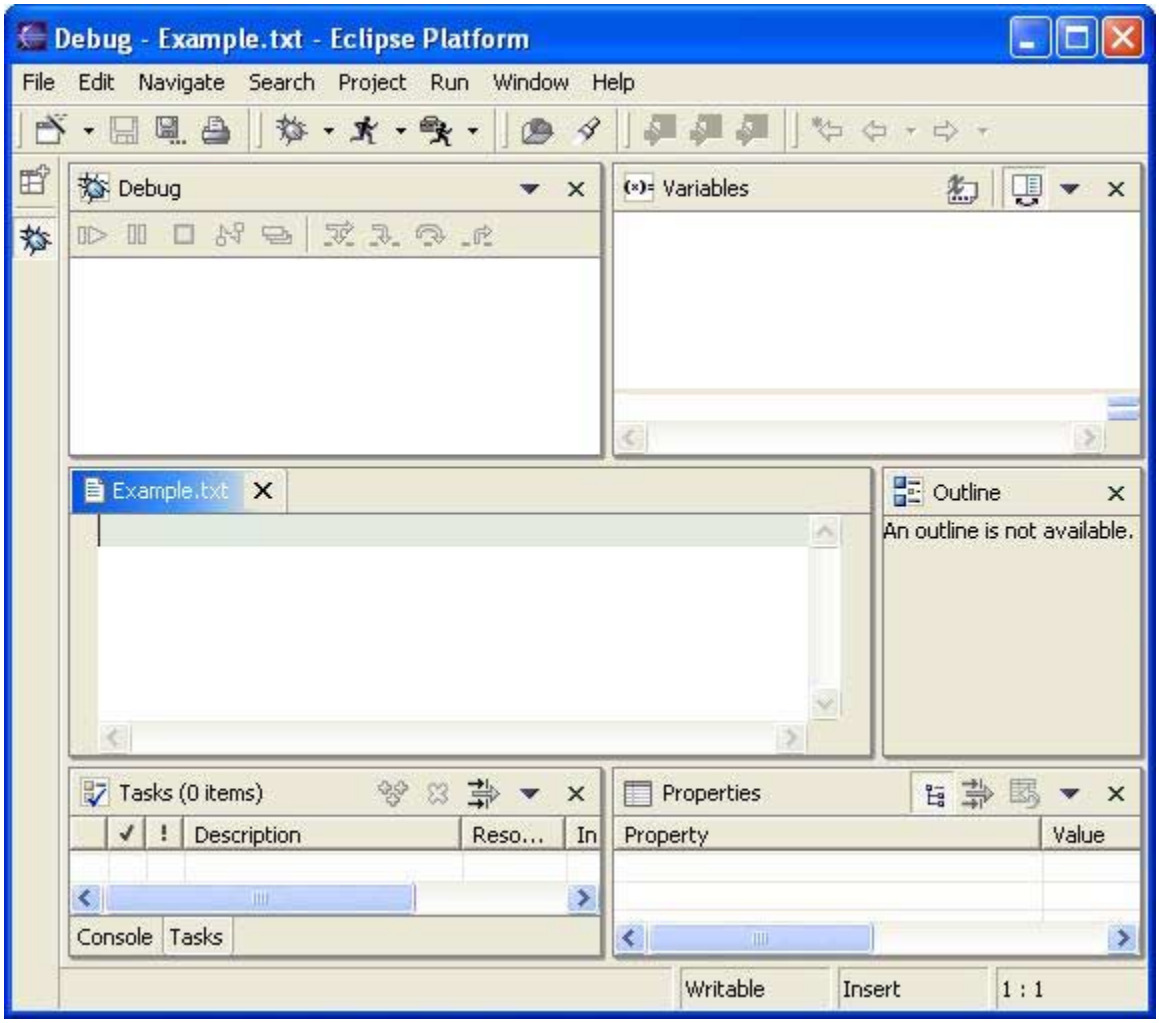


Figure 3-8: An Eclipse Workbench window in the Debug perspective, containing a text editor and the Debug, Variables, Console, and Properties views.

Chapter 4

SDT Common Coding and Debugging Features

The time spent debugging software can take up to 50% of total development time [3] [14]. The SDT seeks to reduce this time through the features described in this chapter. First, program creation and code editing facilities are described (4.1). Then, breakpoint functionality is presented (4.2), followed by program launching and compilation support (4.3). Finally, general debugging (4.4) and help (4.5) support are detailed.

4.1 Code Editing

The SDT adopts Eclipse's resource management system of projects and files by associating each StreamIt application with its own StreamIt project. A project wizard automatically links each newly created project with the SteamIt compiler, while StreamIt source files are created in the same way as text files (Figures 4-1 and 4-2). Users can perform operations on these projects and their files in the Navigator view like any other project in Eclipse.

The StreamIt perspective groups menus, toolbars, and views related to development of StreamIt applications: the StreamIt menu and the menu items for StreamIt launching (described below). The StreamIt Editor is a graphical text editor that provides

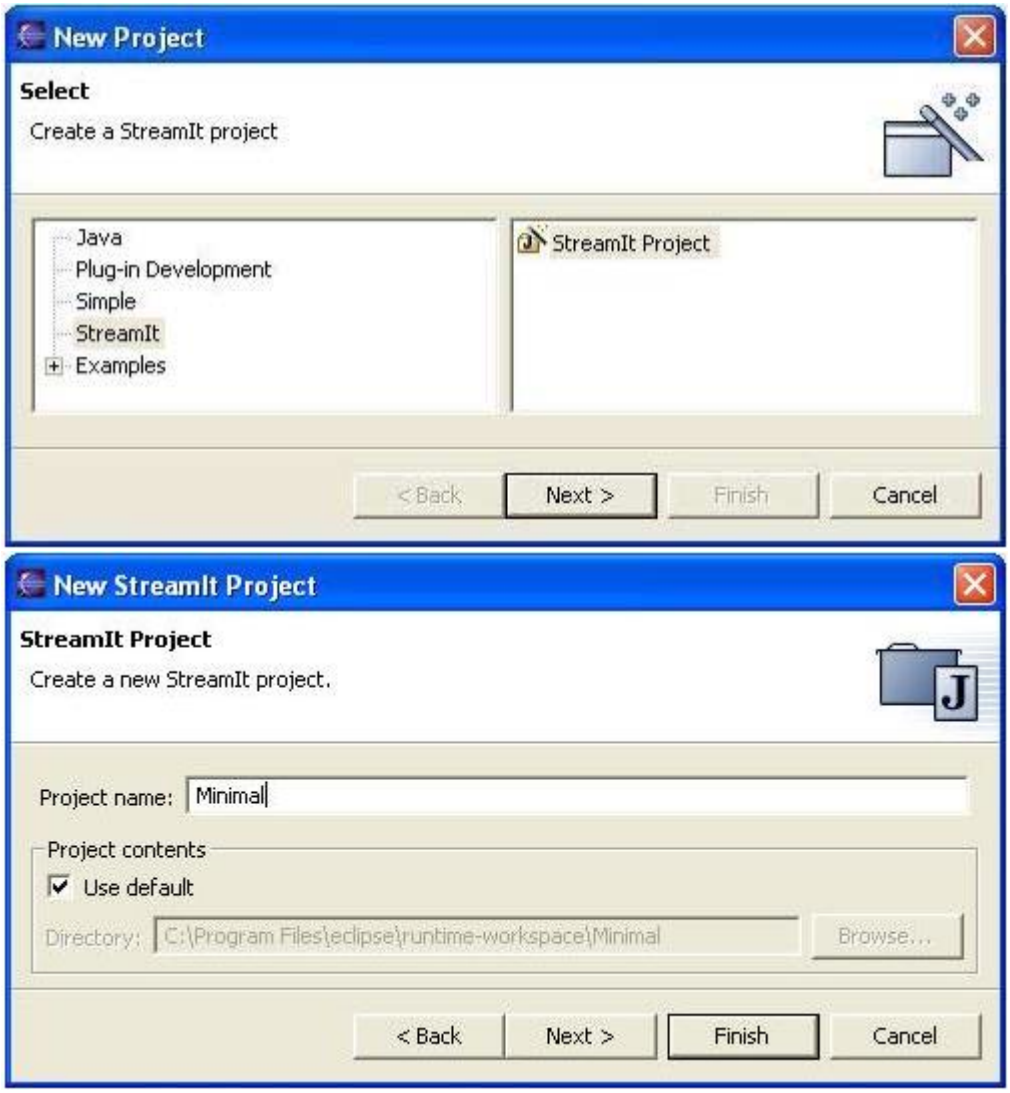


Figure 4-1: A StreamIt project wizard.

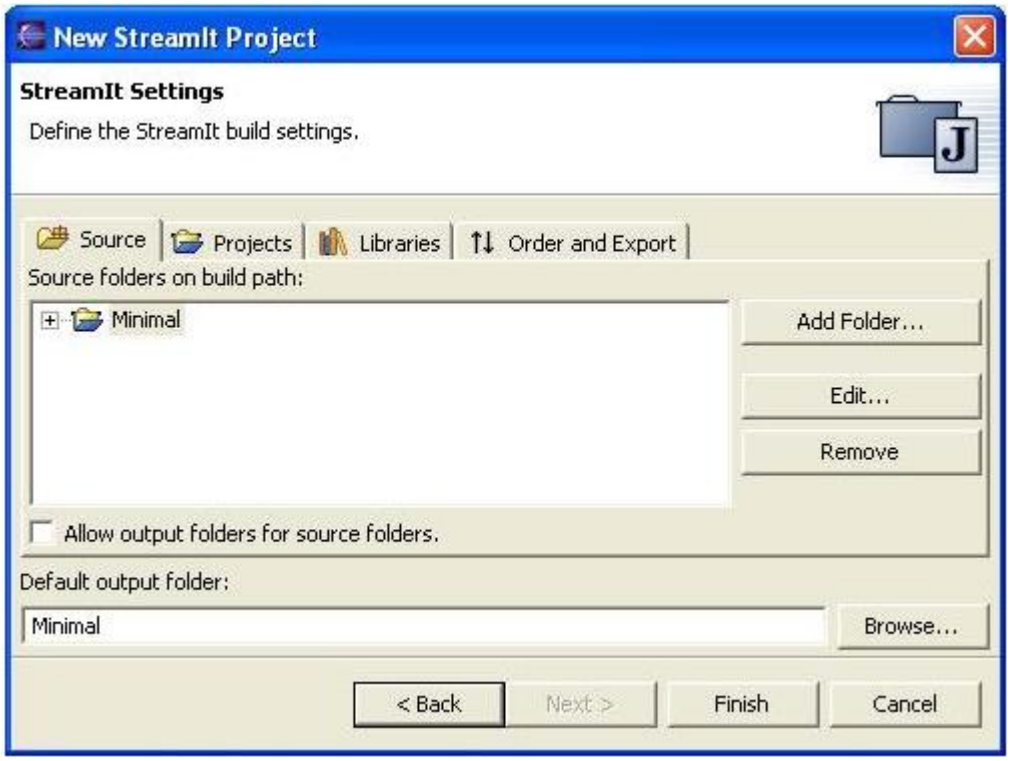


Figure 4-2: A StreamIt project wizard (continued).

default syntax highlighting and basic code indenting, in addition to the functionality provided by Eclipse's text editor (Figure 4-3). When a StreamIt file is open, the Outline view also displays a code-based, hierarchical outline of the file. Each stream declaration is represented as a parent node, with its child nodes being state variables or relevant functions (e.g., `init`, `work`, etc). Parent nodes can be expanded and collapsed by clicking on plus and minus boxes to respectively reveal and hide information as the user chooses (Figures 4-4(a) and 4-4(b)). Each node shows the name and type (if applicable) of its stream, state variable, or function. Furthermore, each node displays an icon from Figure 4-4 depending on what it is representing. In order to maintain look-and-feel and icon-idea association, state variable and method icons are leveraged from icons for package-visible fields and public methods created by Eclipse's JDT. However, because no analogous concept in Java exists, the icons for streams are selected from Eclipse based on shape or lettering, as seen in Figure 4-4. A downward pointing triangle indicates a filter, a long rectangle a pipeline, a circle

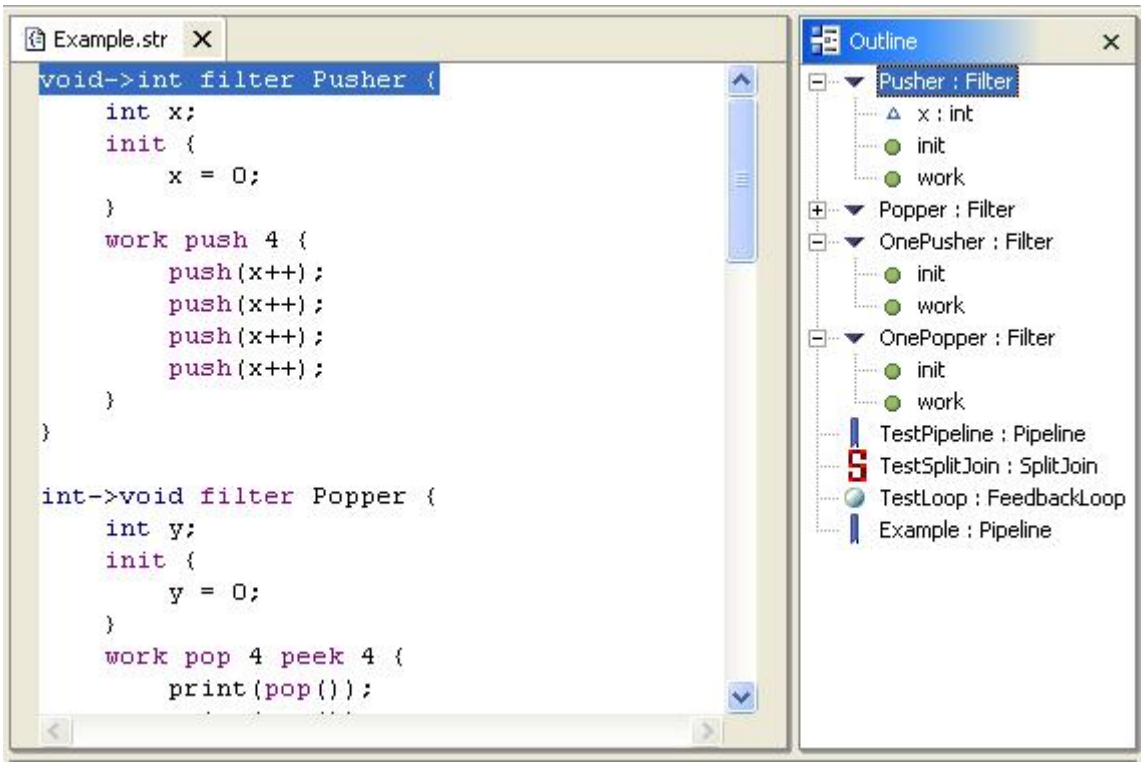


Figure 4-3: A StreamIt Editor with an associated Outline view.

a feedback-loop, and an S a split-join. The Outline view and the StreamIt Editor are linked together such that clicking on a node in the former causes the editor to scroll to the corresponding code and scrolling through the latter results in the Outline view highlighting a node associated with the currently displayed code.

A preference page is available for customizing the syntax highlighting for the StreamIt language (Figure 4-5). Keywords are grouped into categories which define the particular highlight color, and can be added and deleted as desired.

(a) (b) (c) (d) (e) (f) (g) (h)

Figure 4-4: Icons used by the Outline view for StreamIt files: a plus box (a), a minus box (b), a state variable (c), a method (d), a filter (e), a pipeline (f), a split-join (g), and a feedback-loop (h).

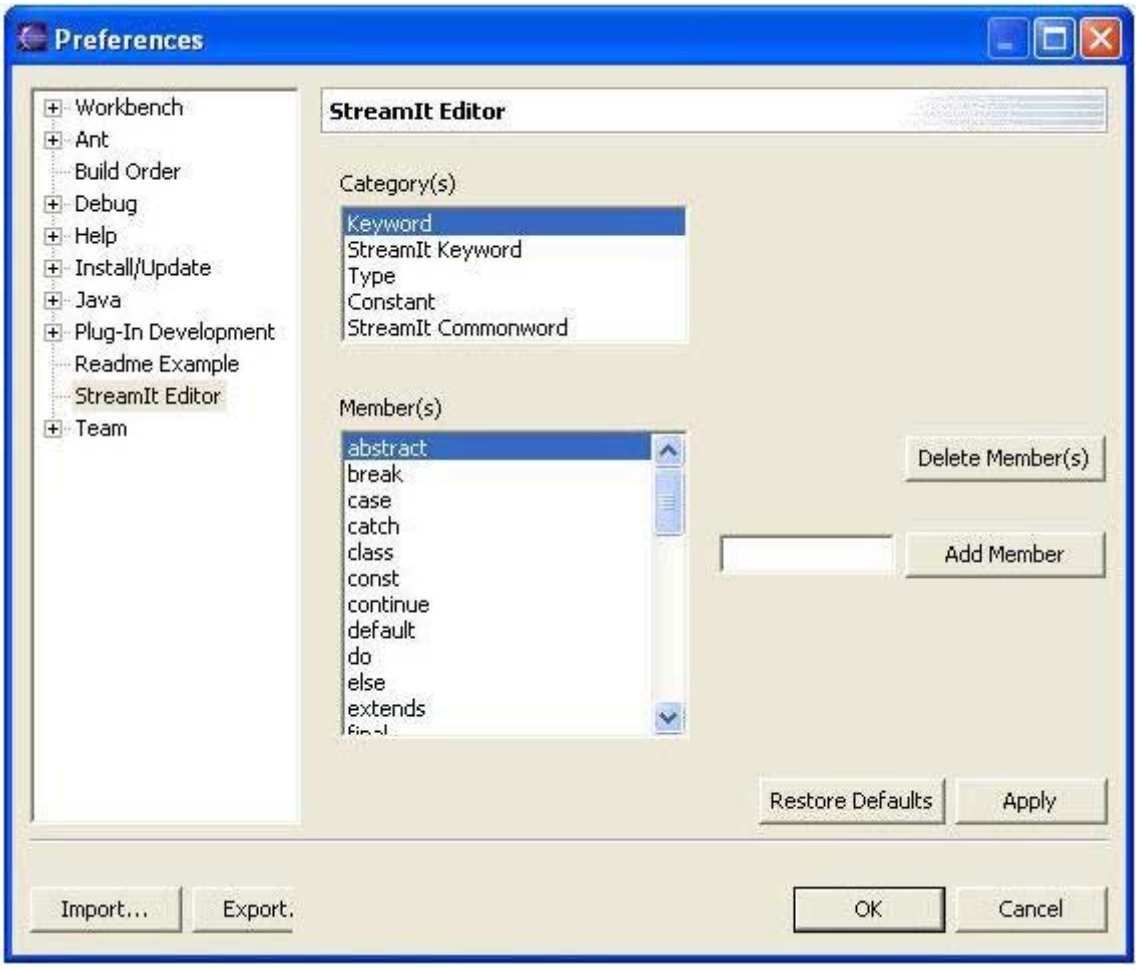


Figure 4-5: A preference page for adding and deleting StreamIt keywords.

4.2 Breakpoints

Breakpoints and watchpoints are the means by which StreamIt programmers can pause an executing StreamIt application at a particular place in code. This corresponds to pausing execution when certain parts of the stream graph and code are reached. Many functionality decisions related to performing operations, such as adding, removing, enabling, etc, on breakpoints and watchpoints were influenced by mirroring Eclipse JDT conventions for Java breakpoints and being externally consistent.

4.2.1 Line Breakpoints

The left gutter of a StreamIt Editor containing StreamIt code (Figure 4-6(a)) is used to add and remove line breakpoints, method breakpoints, and watchpoints. Line breakpoints—breakpoints that cause an application to pause right before a certain line of the code is executed—are the most general purpose of the three. Line breakpoints can be added in four ways:

(1) double-clicking on the left gutter at a location horizontal to a line of code,

(2) right-clicking on the left gutter and selecting "Add Breakpoint" from the pop-up menu that appears under the mouse (Figure 4-6(a)),

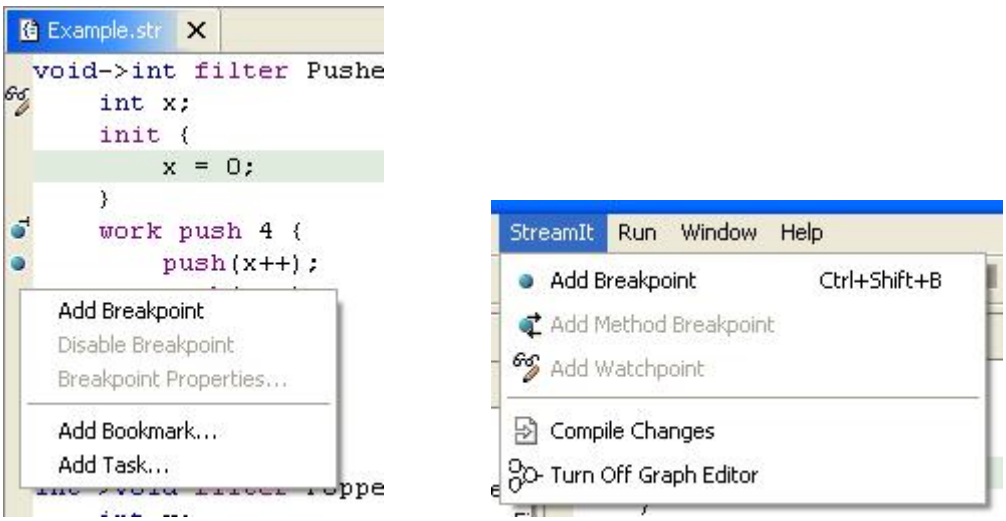
(3) holding the keyboard's `Ctrl` and `Shift` keys and then pressing the `B` key when the StreamIt Editor's cursor is at a line of code, or

(4) selecting "Add Breakpoint" from the StreamIt drop-down menu found in the Workbench's main menu bar (Figure 4-6(b)) when the StreamIt Editor's cursor is at a line of code of interest.

These four options are the same as those available for Java line breakpoints, as is the icon chosen for line breakpoints (Figure 4-7).

Likewise, four corresponding ways are available for removing line breakpoints:

(1) double-clicking on the left gutter where a line breakpoint is,

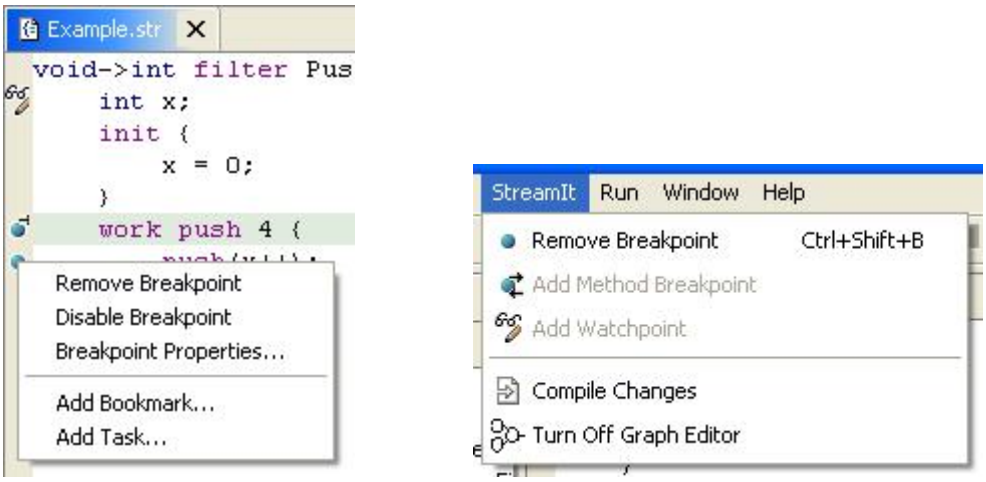


(a) (b)

Figure 4-6: A StreamIt Editor's left gutter showing line breakpoints, method breakpoints, and watchpoints that have been added (a). A pop-up menu allows users to add line breakpoints. The StreamIt drop-down menu found in the Workbench's main menu bar also allows users to add line breakpoints (b).

(a) (b) (c)

Figure 4-7: Breakpoint icons used in the StreamIt Editor and StreamIt drop-down menu for line breakpoints (a), method breakpoints (b), and watchpoints (c).



(a) (b)

Figure 4-8: A pop-up menu allows users to remove line breakpoints (a). The StreamIt drop-down menu allows users to remove line breakpoints (b).

(2) right-clicking on the left gutter and selecting "Remove Breakpoint" from the pop-up menu that appears under the mouse (Figure 4-8(a)),

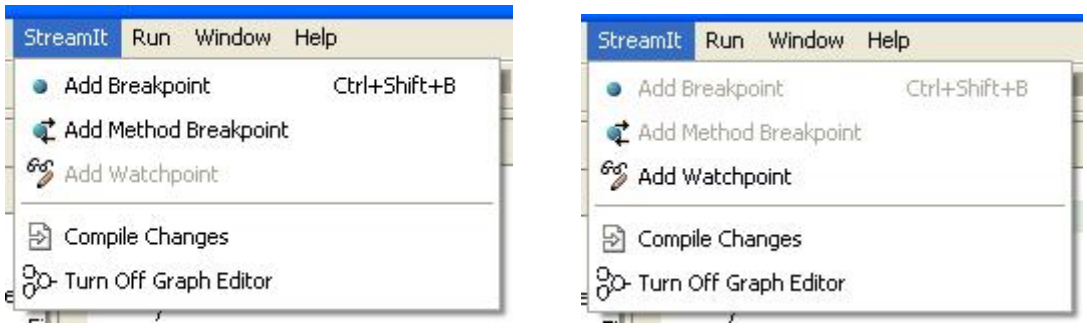
(3) holding the keyboard's `Ctrl` and `Shift` keys and then pressing the `B` key when the StreamIt Editor's cursor is at a line of code that has a line breakpoint, and

(4) selecting "Remove Breakpoint" from the StreamIt drop-down menu (Figure 4-8(b)) when the StreamIt Editor's cursor is at a line of code that has a line breakpoint.

These four ways are the same ways that Java line breakpoints can be removed.

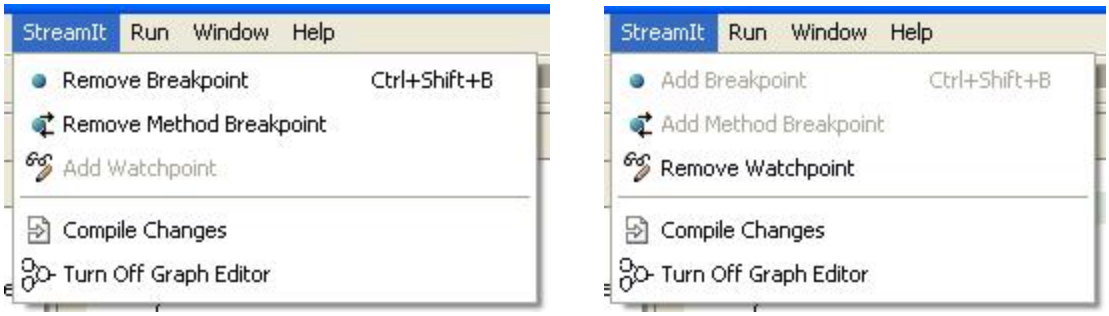
4.2.2 Method Breakpoints and Watchpoints

Method breakpoints cause an application to pause right before or after a certain method is executed and watchpoints cause an application to pause when a variable is modified or accessed. Because both are specialized versions of line breakpoints and are therefore less used, only one way is provided for adding method breakpoints and watchpoint—selecting "Add Method Breakpoint" or "Add Watchpoint" from the



(a) (b)

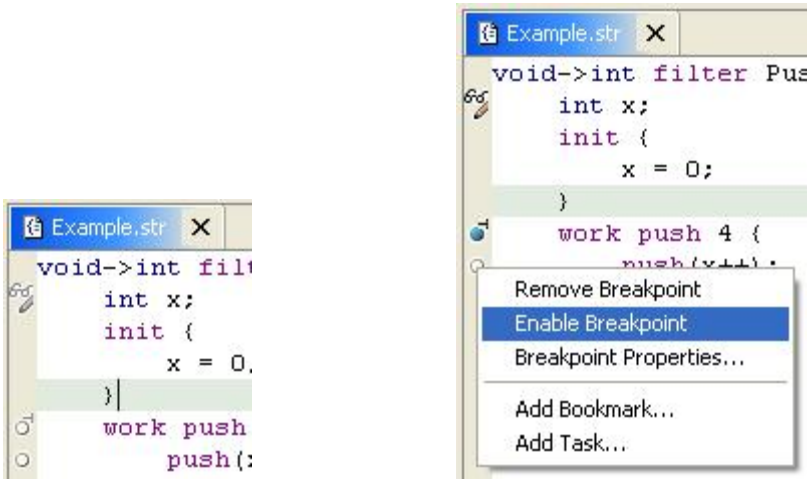
Figure 4-9: The StreamIt drop-down menu allows the user to add method breakpoints (a) and watchpoints (b).



(a) (b)

Figure 4-10: The StreamIt drop-down menu allows the user to remove method breakpoints (a) and watchpoints (b).

StreamIt drop-down menu (Figure 4-9). Method breakpoints can be removed in three ways, which correspond to (2), (3), and (4) for removing a line breakpoint (Figure 4-10). Watchpoints can be removed in two ways, which correspond to (2) and (4) for removing a line breakpoint (Figure 4-10). Again, this functionality mirrors the way that Java method breakpoints and watchpoints are added and removed, as are the icons chosen (Figure 4-7).



(a) (b)

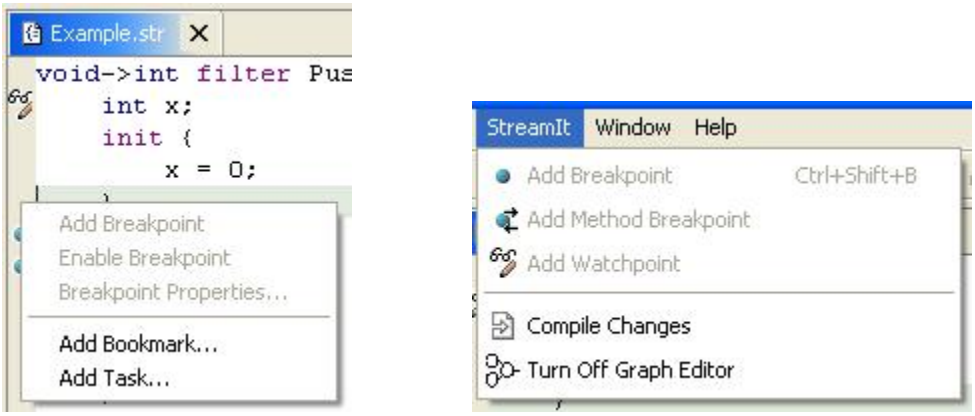
Figure 4-11: Disabled watchpoint, method breakpoint, and line breakpoints (a). A pop-up menu allows users to disable or enable breakpoints or watchpoint (b).

4.2.3 Breakpoint Property Specification

Disabling a breakpoint or watchpoint means that an application being debugged behaves as if that particular breakpoint or watchpoint does not exist. Instead of frequently adding and removing the same breakpoint, disabling allows a programmer to remember that a breakpoint was added at a particular location and rendered inactive until later. Breakpoints and watchpoints are enabled and disabled by right-clicking on the left gutter and selecting "Disable Breakpoint" or "Enable Breakpoint" from the pop-up menu that appears under the mouse (Figure 4-11).

StreamIt breakpoints and watchpoints also inherit conditional breaking functionality from Java breakpoints. That is, StreamIt line breakpoints can be configured to suspend programs only when an expression is true or when its value changes, while method breakpoints and watchpoints can be configured to suspend on either entry or exit and either access or modification, respectively. For more information on configuring breakpoint properties, see [11].

In general, line breakpoints cannot be added when the associated line contains no executable code (such as a line with only an ending bracket), while method break-



(a) (b)

Figure 4-12: Disabled selections in the StreamIt Editor's pop-up menu (a) and StreamIt drop-down menu (b).

points and watchpoints cannot be added on non-methods or non-variables. Users are prevented from adding breakpoints or watchpoints at such invalid lines of code by disabling the selections in the StreamIt drop-down menu and the StreamIt Editor's pop-up menu (Figure 4-12). Furthermore, hot key codes and double-clicking are not responsive at invalid lines.

4.3 Program Compilation and Launch Support

StreamIt applications need to be compiled whenever the user would like to add breakpoints or discover syntactic errors. Compiling the file currently being displayed in the StreamIt Editor can be done by selecting "Compile Changes" from the StreamIt drop-down menu (Figure 4-13).

Syntax errors and compilation warnings and errors from the StreamIt compiler are reported in several ways, as depicted in Figure 4-14:

- In the Tasks view, the warnings and errors are listed by a short textual description of the problem, an icon indicating severity (i.e. warning or error), the name of the associated file and project, and its line number.

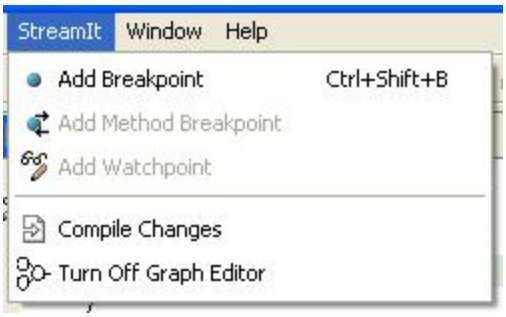


Figure 4-13: The StreamIt drop-down menu allows the user to compile the file currently being displayed in the StreamIt Editor.

- In the Package Explorer view, severity icons appear on the affected StreamIt file.
- In the StreamIt Editor's left gutter and on its tab, severity icons are displayed near the affected line.
- Squiggly lines appear under the text that might be causing the problem.

The problem description also appears when the mouse is hovered over the severity icon in the left gutter. In the Tasks view, right clicking on a problem and selecting "Go To" from the pop-up menu that appears under the mouse opens the associated file at the location of the problem.

The SDT also assumes Eclipse's conventions for launching programs and configuring the parameters of launches. Programs are launched in several ways, depending on whether the user wishes to run or debug his or her application (Figure 4-15):

- selecting "Run As" or "Debug As" and then "StreamIt Application" from the Run drop-down menu found in the Workbench's main menu bar when a StreamIt file is open or selected in the Package Explorer view,
- clicking on the run or debug icons from the Workbench's main toolbar and selecting "Run As" or "Debug As" and then "StreamIt Application" from the drop-down menu that appears when a StreamIt file is open or selected in the Package Explorer view,

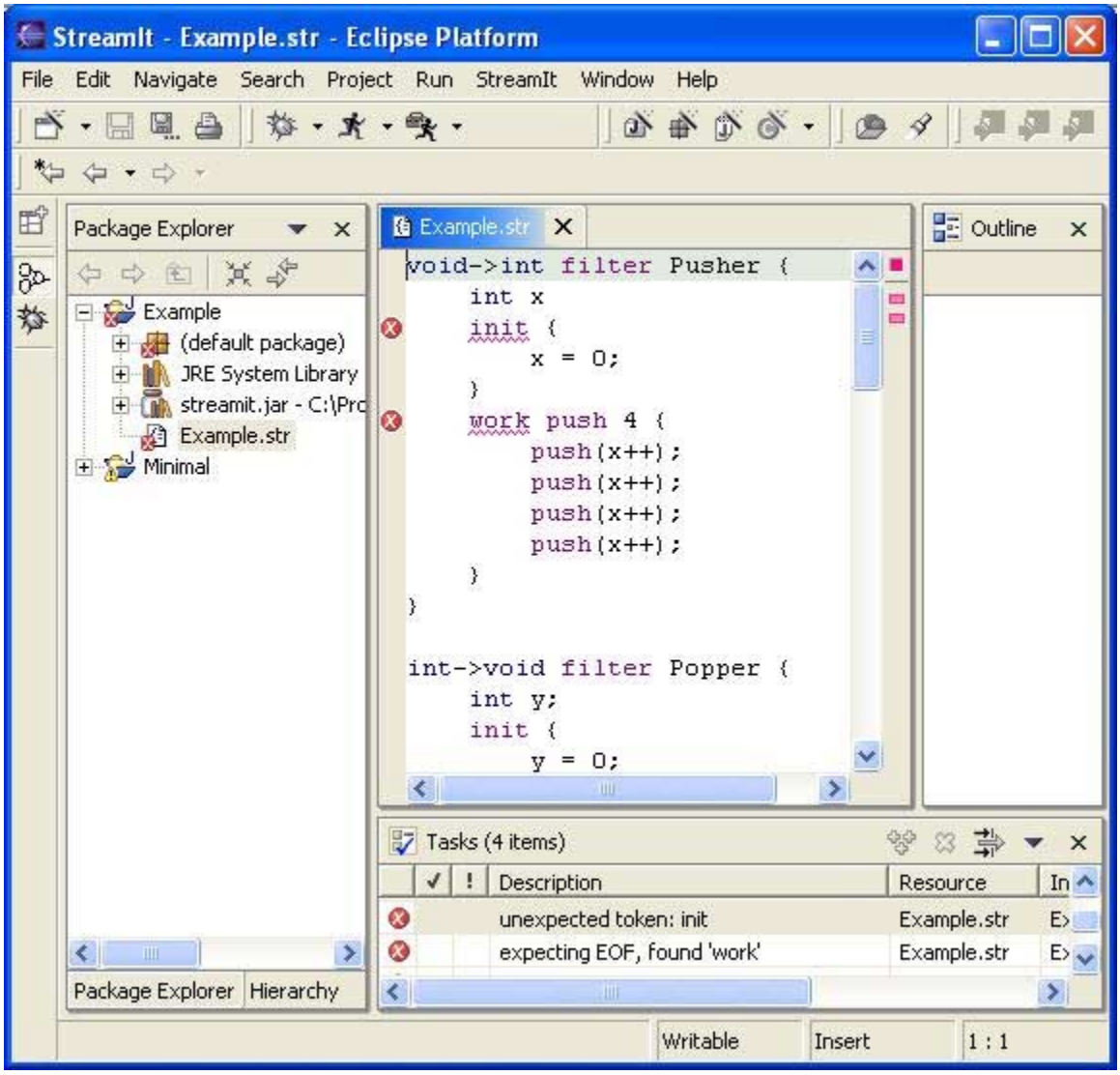


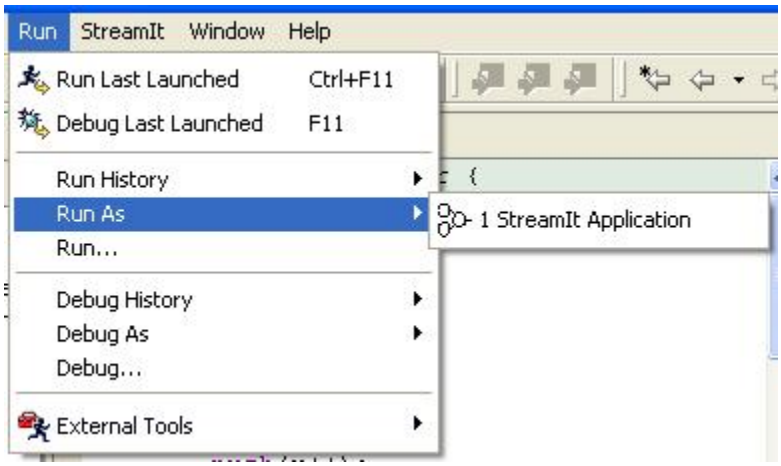
Figure 4-14: Errors and warnings are displayed in the Tasks view, Package Explorer view, and StreamIt Editor.

- selecting “Run History” or “Debug History” and then a previously launched application from the Run drop-down menu,
- clicking on the main toolbar’s run or debug icons and then selecting a previously launched application from the drop-down menu that appears,
- selecting “Run Last Launched” or “Debug Last Launched” from the Run drop-down menu,
- selecting “Run ” or “Debug ” from the Run drop-down menu, or
- clicking on the main toolbar’s run or debug icons and then selecting “Run ” or “Debug ” from the drop-down menu that appears.

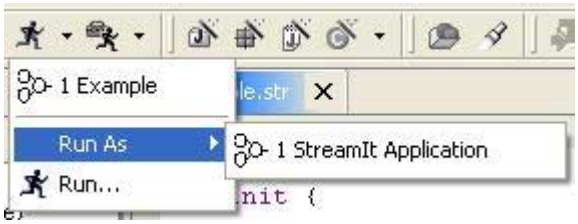
For the first two methods, an application is run or debugged with a default configuration. For the last two methods, a launch configuration management wizard appears that enables programmers to create and configure several different launches for a particular application (Figures 4-16 and 4-17). Launches can be configured for the number of iterations of the top-level pipeline or secondary StreamIt files to include in a launch (Figure 4-18). Secondary files are useful when large amounts of code are split among several files in a modular way.

4.4 General Debugging Support

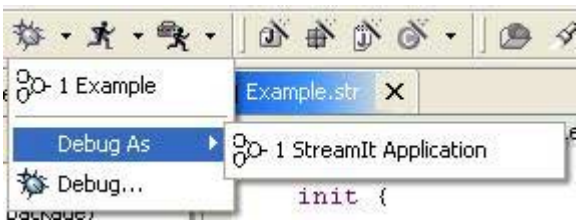
When StreamIt program is launched, the Debug perspective displays code in the StreamIt Editor and five views of interest to the user: the Debug, Variables, Console, Stream Graph, and Overview of Stream Graph views (Figure 4-19). The Console view prints anything written to standard-out or standard-error and sends anything to be read to standard-in. The Stream Graph and Overview of Stream Graph views are described in chapter 5. In the Debug view, each launch is registered as a parent node with its children as sub-processes or sub-threads, which in turn have child stack frames. In general, the main thread with its two stack frames is the only thread of interest to a StreamIt programmer. One stack frame corresponds to the stream whose `init`



(a)

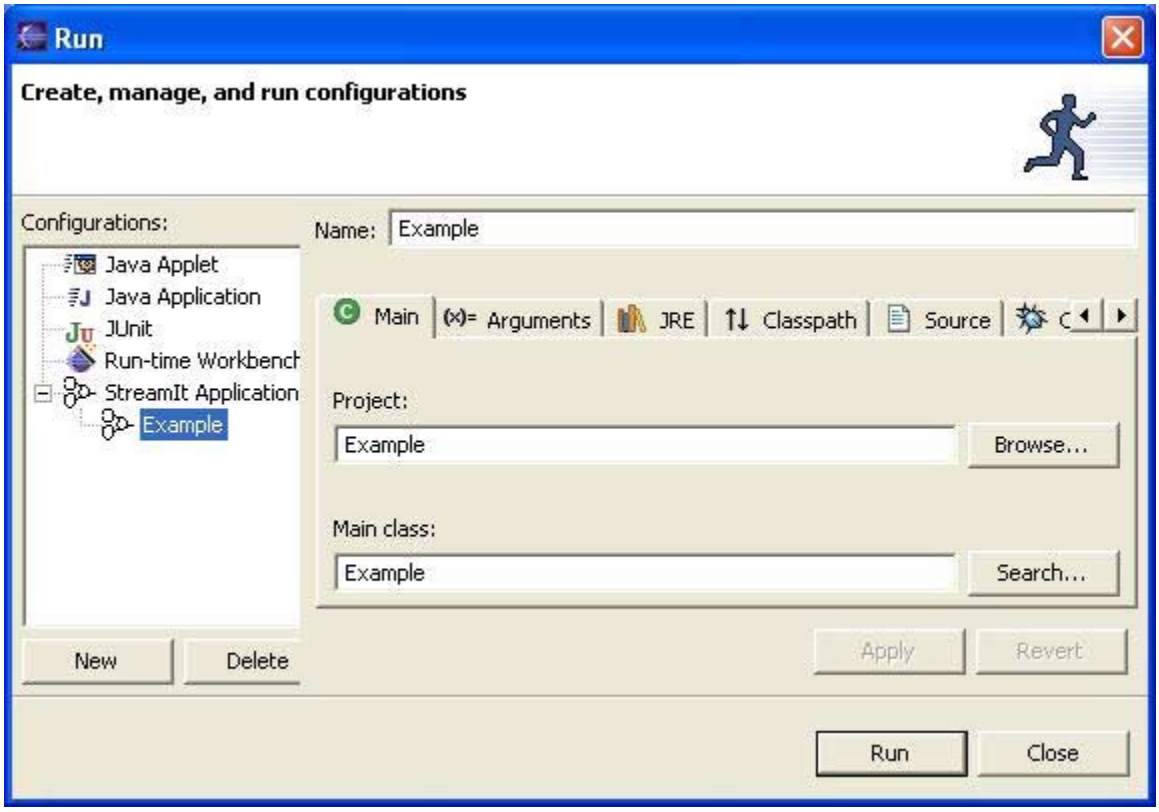


(b)



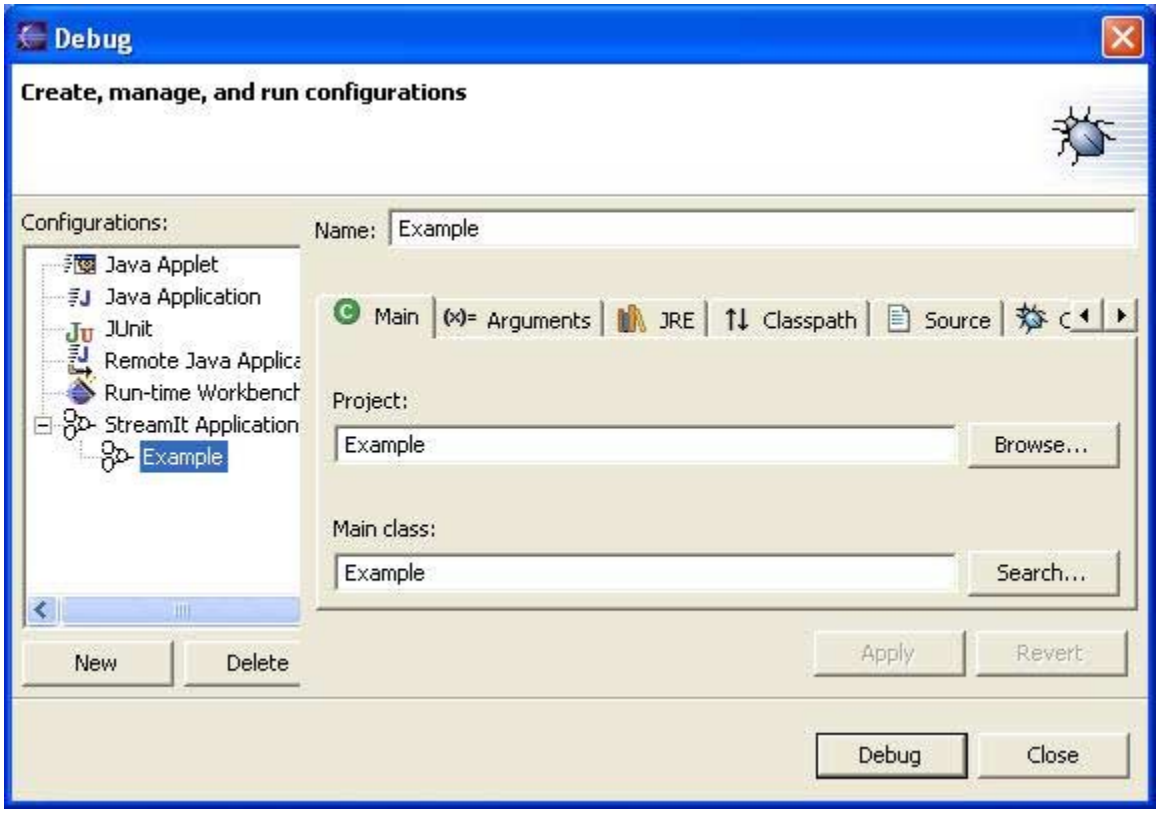
(c)

Figure 4-15: The Run drop-down menu (a) and the toolbar's run (b) and debug (c) icons allow users to launch StreamIt applications.



(a)

Figure 4-16: Launch configuration management wizard for running.



(a)

Figure 4-17: Launch configuration management wizard for debugging.

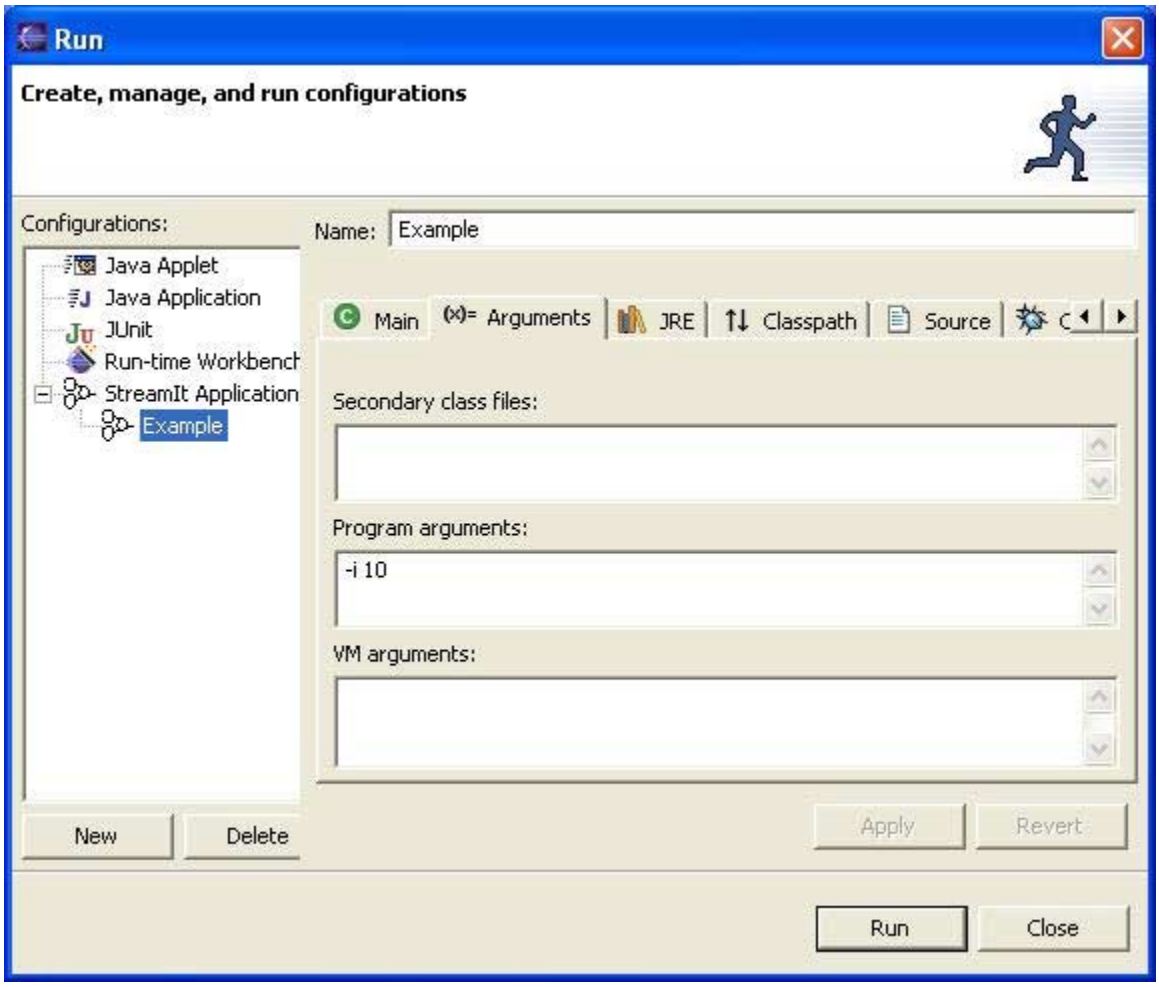


Figure 4-18: The arguments tabs allows user to specify iterations or secondary StreamIt files to include in a launch.

or `work` function is currently executing, while the other frame corresponds to the top-level pipeline. When either stack frame is selected, the Variables view displays a stream as a parent node whose child nodes are either state variables or child streams. Stream nodes show a name and instance id number, while state variable nodes display a name, type, and value. Like nodes in the Outline view, nodes here can be expanded and collapsed by clicking on plus and minus boxes to reflect the hierarchical nature of stream programs.

In addition to the Variables view, user selection of a stack frame in the Debug view propagate to the Stream Editor, highlighting the associated stream declaration, and the Stream Graph and Overview of Stream Graph views, highlighting the associated stream in the graph. This feature is intended to (1) show what is currently being executing, (2) convey the correspondence between the views and editor, and (3) offer another way for the user to navigate a stream graph. Representing highlights as a blue background with white letters is externally consistent with other editors and views in the Eclipse platform. Furthermore, because all launches are registered with the Debug view, user selections of different launches will result in the Stream Graph and Overview of Stream Graph views changing to show the stream graph associated with currently selected launch.

A StreamIt application runs until a breakpoint or watchpoint is reached, at which time execution is suspended. The editor and views (except for the Console view) highlight their representation of the currently executing part of the program, as seen in Figure 4-19. For the StreamIt Editor, the line of code about to be executed is automatically opened, scrolled to, and highlighted if applicable. As previously mentioned, the Debug view highlights the stack frame whose stream's `init` or `work` function is executing, while the Variables view displays the stream, its child streams, and its state variables. The Stream Graph and Overview of Stream Graph views also highlight and scroll to the stream (as described in chapter 5).

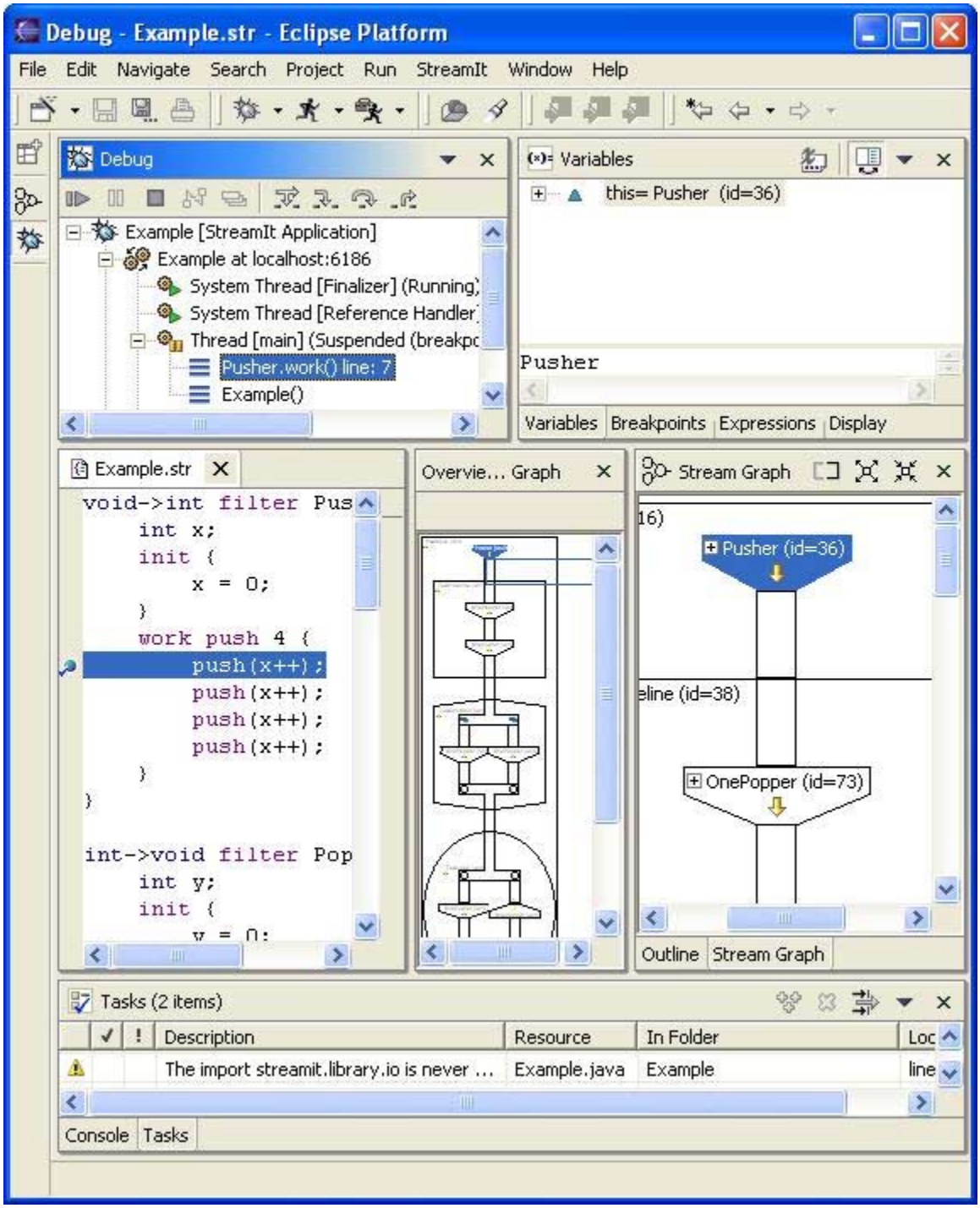


Figure 4-19: A StreamIt program suspended at a breakpoint in the Debug perspective.

(a) (b) (c)

Figure 4-20: Icons from the Debug view's toolbar and pop-up menu for resuming (a), terminating (b), and stepping over (c).

4.4.1 Features in the Debug View

The Debug view enables StreamIt programmers to resume, terminate, and perform breakpoint stepping of an application. Clicking the resume icon in the Debug view's toolbar, pressing the F8 key, or right-clicking on a suspended thread and selecting "Resume" from the pop-up menu that appears allows a program to run until the next breakpoint or watchpoint is encountered or until completion. Clicking the terminate icon or right-clicking on a suspended thread and selecting "Terminate" from the pop-up menu that appears terminates a program. Clicking the step over icon or pressing the F6 key resumes execution at the line of code originally highlighted when the program was suspended (Figure 4-20). The application is suspended on the next executable line. Stepping can continue until program completion.

Finally, re-launching a StreamIt application is done by right-clicking on a launch and selecting "Relaunch" from the pop-up menu that appears (Figure 4-21).

4.4.2 Variable Value Modification

In the Variables view, the values of state variables can be changed by right-clicking on the variable and selecting "Change Variable Value" from the pop-up menu that appears (Figure 4-22). A "Set Variable Value" dialog provides a text box for entering a new value and detects invalid variable values.

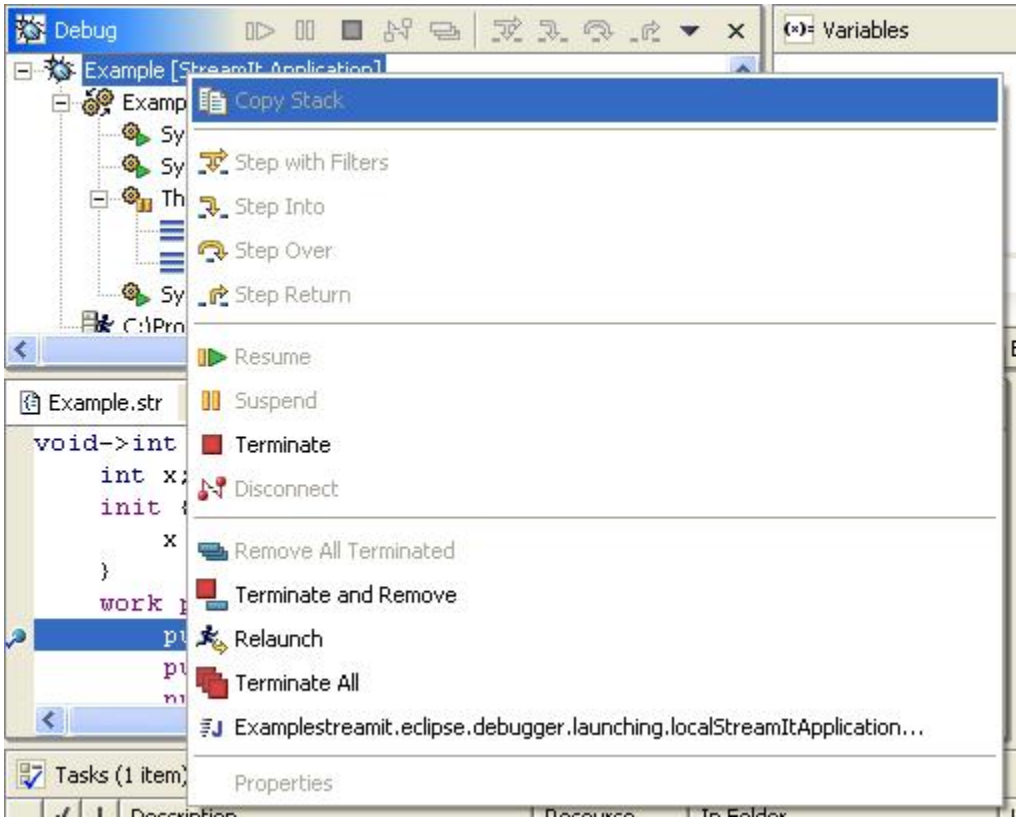
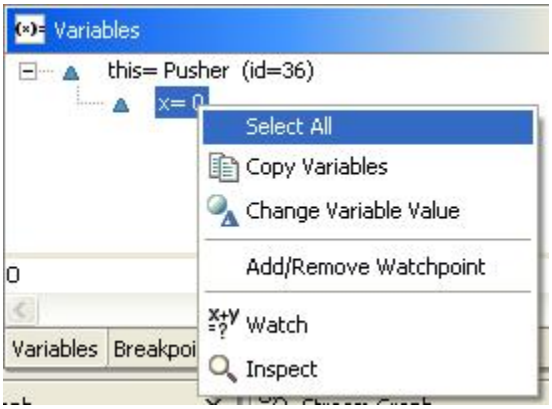
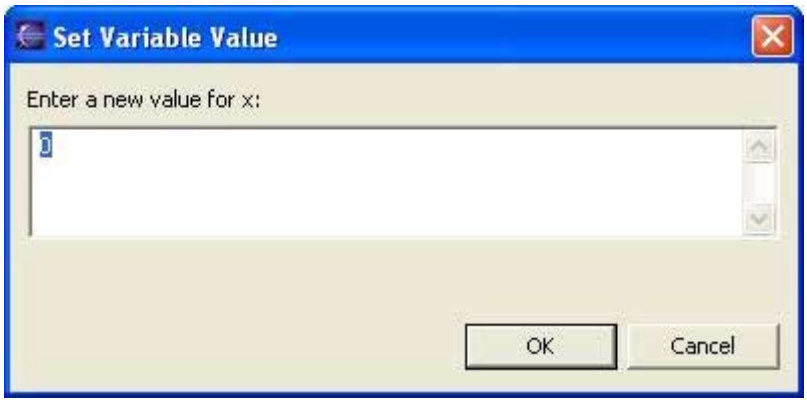


Figure 4-21: The Debug view logs all launches, which allows them to be re-launched.



(a)



(b)

Figure 4-22: The Variables view's pop-up menu (a) and the "Set Variable Value" dialog (b).

4.5 Help Support

The SDT adds "The StreamIt Development User Guide" to Eclipse's help system. The SDT guide consists of instructions for installing Eclipse and the SDT, a basic tutorial on using the SDT, conceptual descriptions of the SDT's features, a section devoted to tasks the user can perform, reference material, and sample StreamIt programs. This guide is accessible in the same way that all Eclipse help manuals are—through the Help menu of the Workbench's main menu bar.

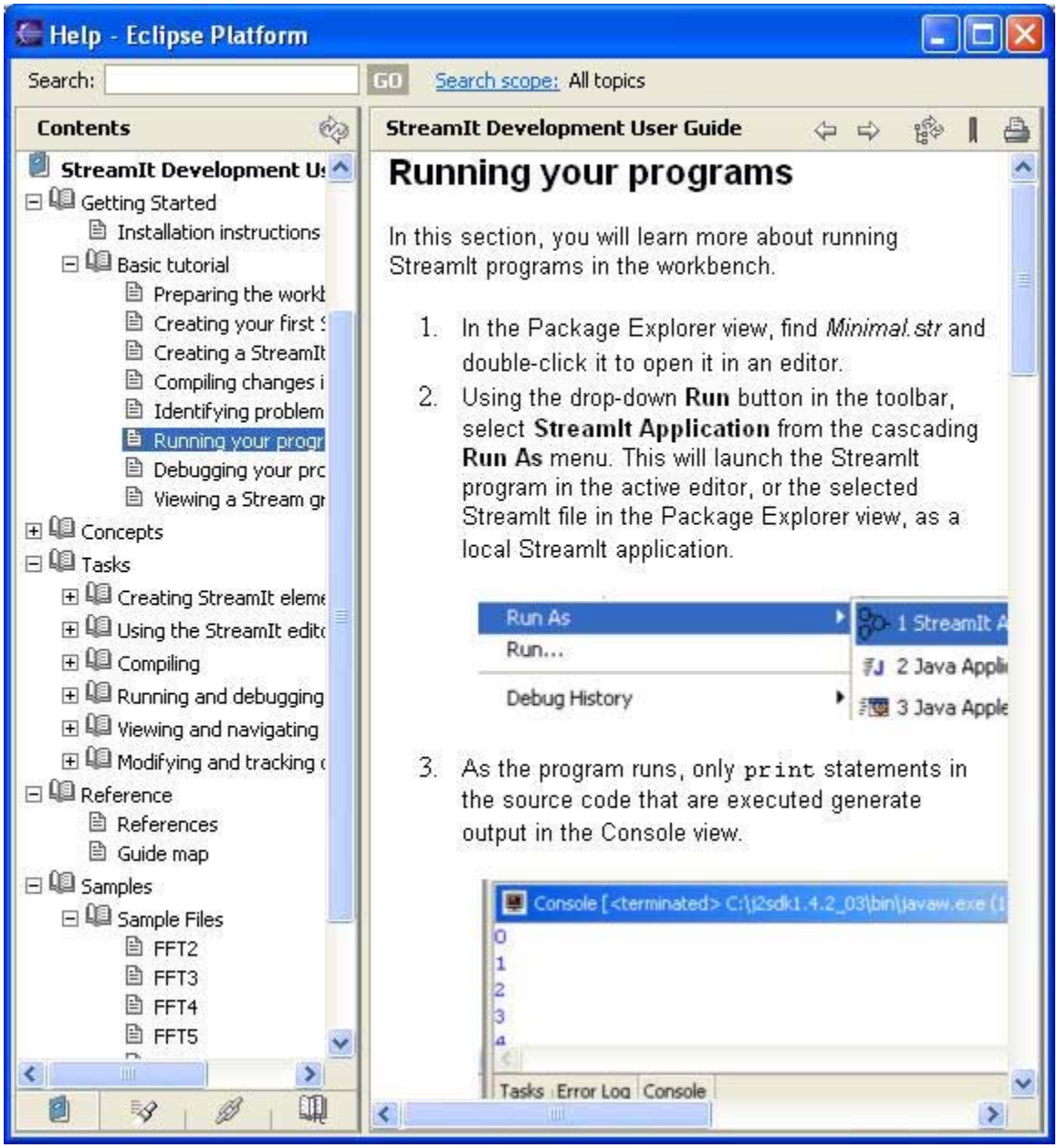


Figure 4-23: The StreamIt Development User Guide.

Chapter 5

SDT Stream Graph-Specific Features

This chapter is divided into two sections directed at StreamIt-specific debugging and visualization: stream graph examination and hierarchical navigation (5.1) and data display, modification, and tracking (5.2).

5.1 Stream Graph Examination and Hierarchical Navigation

When a StreamIt program is suspended at a breakpoint or watchpoint, the program's stream graph representation is displayed in the Stream Graph (Figure 5-1) and Overview of Stream Graph (Figure 5-2) views. The Stream Graph view is the primary, interactive way for a programmer to examine and navigate through the stream graph. In contrast, the Overview of Stream Graph view reflects the contents of the Stream Graph view scaled down 75%. When a program is initially suspended, the entire the stream graph is shown.

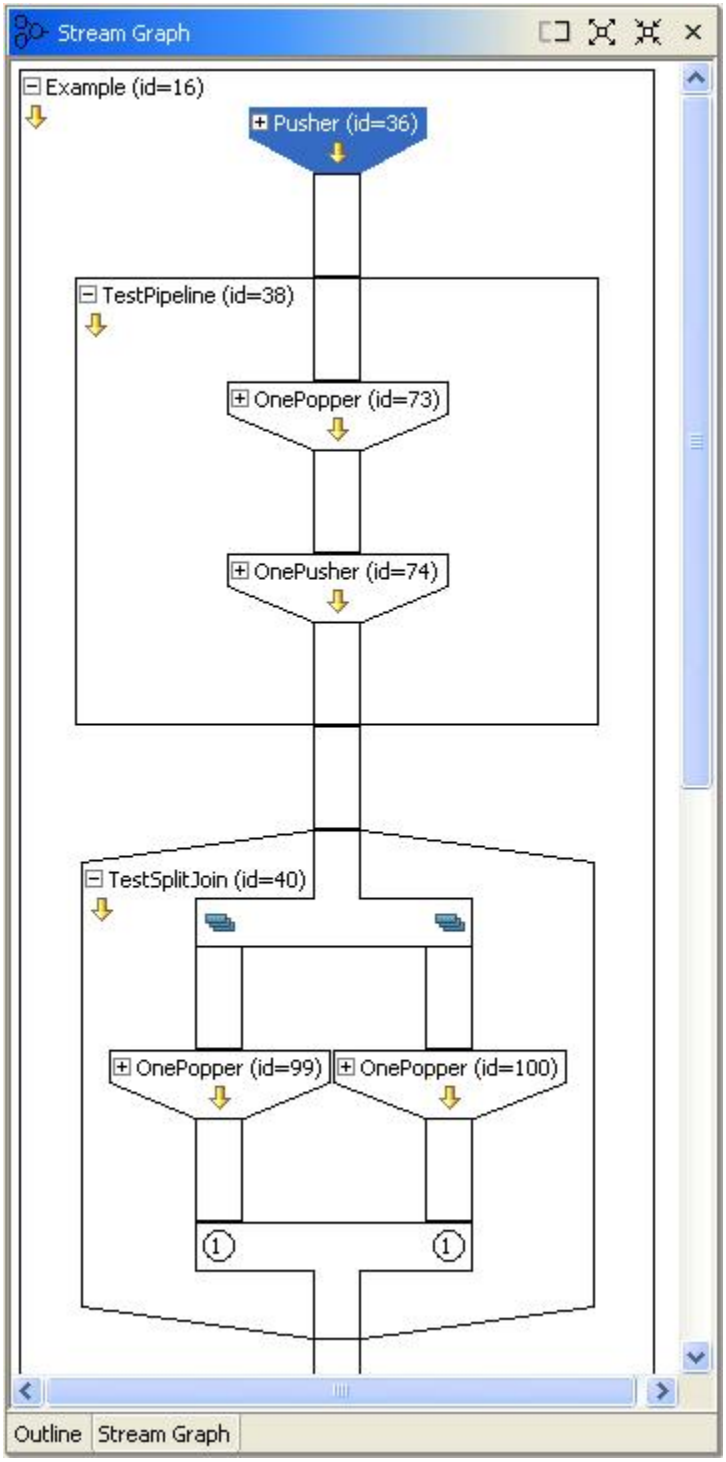


Figure 5-1: The Stream Graph view displaying a stream graph during program suspension.

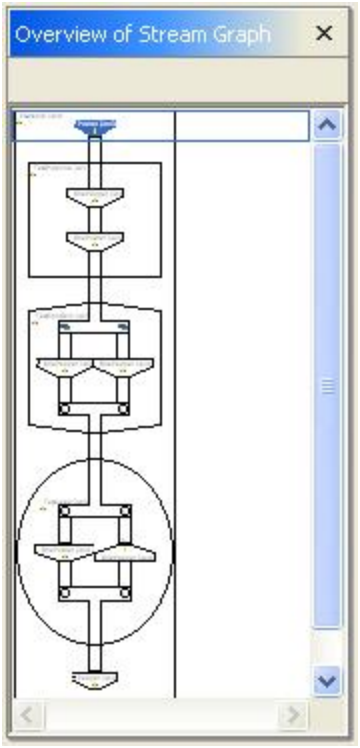
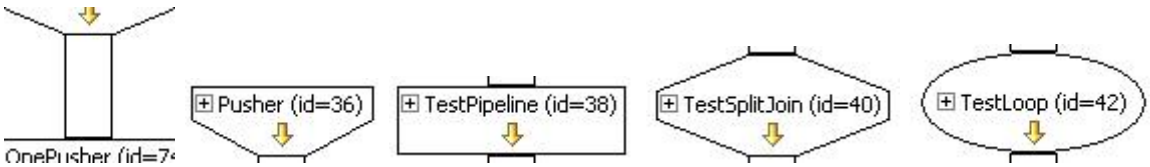


Figure 5-2: The Overview of Stream Graph view displaying a stream graph during program suspension.



(a) (b) (c) (d) (e)

Figure 5-3: The Stream Graph view displays channels (a), filters (b), pipelines (c), split-joins (d), and feedback-loops (e).

5.1.1 Streams in the Stream Graph View

Channels and streams are depicted with different shapes to suggest their meaning. As seen in Figure 5-3, channels are drawn as elongated rectangles to convey that data is only passing through them. In contrast, streams are wider figures to suggest data manipulation: Filters are meant to look like funnels, while pipelines are depicted as wide rectangles, indicating that they can contain more than just data. A split-join is drawn as a blunted diamond, intended to imply that the data within channels are split, operated on, and then joined together, while a feedback-loop is an ellipse to suggest a loop.

A child stream is rendered depending on whether its parent is a pipeline, split-join, or feedback-loop (Figure 5-4). The children of a pipeline are drawn in a column starting from its parent's input channel and ending at its parent's output channel. As a result, the input channels of the pipeline and its first child are connected, and the output channels of the pipeline and its last child are connected. In contrast, a split-join's children are drawn left to right in a row, with its splitter and joiner depicted as T-shaped channels. The splitter is drawn from the parent's input channel to each child's input channel, while the joiner is drawn from the parent's output channel to each child's output channel. The body and loop children of a feedback-loop are also drawn left to right in a row, but the loop child is drawn in the reverse direction of the body child. In contrast to a split-join, the joiner is drawn from the parent's input channel to each child's input channel, while the joiner is drawn from the parent's

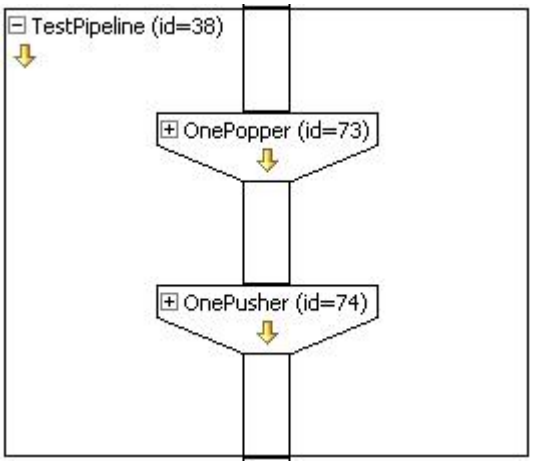
output channel to each child's output channel. The I/O types of all splitters and joiners are indicated with an icon or circle at a location above or below each child's input or output channel, respectively, yet still within the splitter or joiner (Figure 5-4). Duplicate splitters are designated by a duplicate icon, while roundrobin splitters and joiners are specified by a circle containing a number representing the weight given to the child stream above or below it (Figures 5-5(a) and 5-5(b)).

Streams within the Stream Graph view are also portrayed with their name, instance id, and an icon of a yellow arrow (Figures 5-1, 5-5(c), and 5-5(d)). The instance id next to a stream's name corresponds to a unique number in the Variables view. Ids are useful for programs that use the same streams multiple times, as these streams are indistinguishable otherwise. Furthermore, yellow arrow icons are displayed in every stream to indicate which channel is the input or output channel and to convey the direction that data flows within the stream (i.e., the arrow points from input channel to output channel).

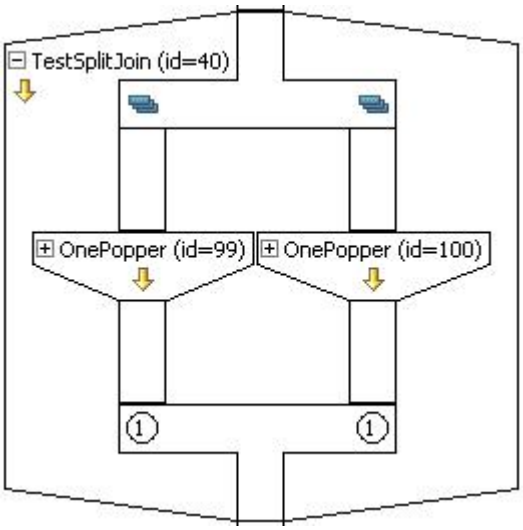
Although the lines of a stream demarcate it from other streams, these lines may become confusing when viewing the graphs of very large StreamIt applications. Therefore, the Stream Graph view provides a "Hide Lines" button within its toolbar (Figures 5-1 and 5-5(e)). Pressing this button hides the lines of any non-filter stream, while depressing this button restores those lines. Therefore, a stream graph with its non-filter lines hidden is only showing a graph of filters connected together.

5.1.2 Stream Graph Debugging Behavior

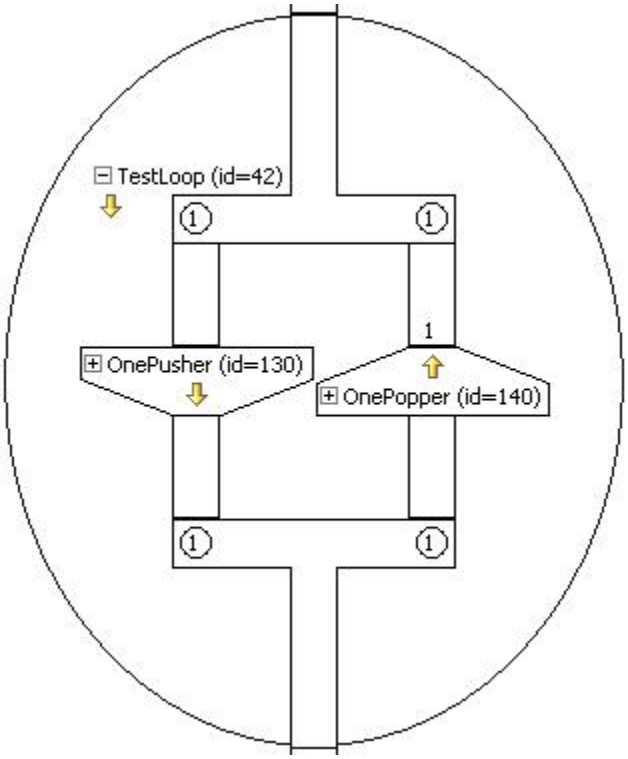
When a StreamIt application is suspended at a breakpoint or watchpoint, the Stream Graph view highlights and scrolls to the stream whose `init` or `work` function is currently being executed. As previously mentioned, because it mirrors the contents of the Stream Graph view, the Overview of Stream Graph view also shows the stream highlighted. Because `init` function build the stream graph, suspending the program during an `init` function means that only the portion of the stream graph that has been created so far is displayed by the Stream Graph view. After a StreamIt program runs to completion, both views continue to display stream graphs so that a program-



(a) A pipeline's children.



(b) A split-join's children.



(c) A feedback-loop's children.

Figure 5-4: The Stream Graph view displays the children of pipelines (a), split-joins (b), and feedback-loops (c).

(a) (b) (c) (d) (e)

Figure 5-5: Stream Graph view icons for duplicate splitters (a), weighted splitters and joiners (b), data flow (c) (d), and "Hide Lines" (e).

(a) (b)

Figure 5-6: The Stream Graph view allows users to add (a) and remove (b) filter instance breakpoints.

mer can reference the stream graph when making any improvements to their code (hopefully based on information learned from the debugging session).

Filter instance breakpoints cause an application to pause right before the `work` function of a particular filter instance is executed. Adding and removing these breakpoints is done by right-clicking on a filter instance within the stream graph and selecting "Add Filter Instance Breakpoint" or "Remove Filter Instance Breakpoint" (depending on the state of the filter instance) from the pop-up menu that appears (Figure 5-6). The same icon used for line breakpoints is used for filter instance breakpoints, as method breakpoints imply that a program suspends on every `work` function of a filter declaration. "Add Filter Instance Breakpoint" and "Remove Filter Instance Breakpoint" are disabled when the program has run to completion.

5.1.3 Stream Graph Navigation

The Stream Graph view offers several ways for a programmer to examine and navigate through various parts of a stream graph, all based on the ability to expand and collapse hierarchical streams (i.e. pipelines, split-joins, and feedback loops).

(a) (b) (c) (d)

Figure 5-7: The Stream Graph view's icons for navigation: a plus box (a), a minus box (b), "Collapse All" (c), and "Expand All" (d).

- Clicking on a plus icon (Figure 5-7(a)) expands a hierarchical stream so that its children are visible (Figure 5-4).
- Clicking on a minus icon (Figure 5-7(b)) collapses a hierarchical stream so that its children are hidden (Figure 5-3).
- Double-clicking on a hierarchical stream toggles its expanded and collapsed state.
- Pressing the "Collapse All" button (Figure 5-7(c)) in the Stream Graph view's toolbar collapses all streams.
- Pressing the "Expand All" button (Figure 5-7(d)) in the Stream Graph view's toolbar expands all streams.

The expanded or collapsed state of the children of an expanded or collapsed hierarchical streams depends on the state the user left them in. For example, take a pipeline P with two child split-joins S and J. Initially, a user expands P and then S, but leaves J collapsed. Next, the user collapses and then expands P. When this happens, S is expanded as well, but J is left collapsed. The state of streams is maintained across program suspensions, although the graph is always expanded to show the currently being executed stream for any suspension. Since large graphs make navigation more difficult, the various ways to navigate the graph are provided for scalability, flexibility, and efficiency.

As a miniaturized version of the Stream Graph view, the Overview of Stream Graph view reflects the contents of the Stream Graph scaled down by 75%, such that everything but size is the same for the sake of consistency. Clicking on a location in the Overview

of Stream Graph view causes the corresponding location in the Stream Graph view to be scrolled to and shown, facilitating faster navigation of large streaming applications. For example, in Figure 5-8, the graph window is too small to show anything of interest. Instead of repeatedly scrolling to the bottom of the graph, the user can make one click to get there. A blue rectangle in the Overview of Stream Graph view indicates what is currently being displayed in the Stream Graph view. The Overview of Stream Graph view is a separate view that can be optionally closed, which may be desirable for small graphs.

5.1.4 Stream Statistics

During program suspension and after program completion, statistics on streams can be found in the Stream Graph and Properties views. In the former view, when filters are expanded using the plus icon previously mentioned (Figure 5-7), programmers can examine (Figure 5-9):

(1) a filter's input and output channels' data types,

(2) a filter's push, pop, and peek rates,

(3) the amount of data pushed and popped so far within a filter's `work` function,

(4) the maximum index of data peeked so far within a filter's `work` function, and

(5) the number of times the `init`, `prework`, or `work` functions have been executed.

In the Properties view, streams are displayed as parent nodes whose child nodes are either properties or child streams (Figure 5-10). All stream nodes have properties corresponding to filter statistics (1) through (4) in the Stream Graph view and a property describing the type of stream, while filter nodes also have (5). Like nodes in the Variables and Outline views, nodes here can be expanded and collapsed by clicking on plus and minus boxes (Figure 5-7).

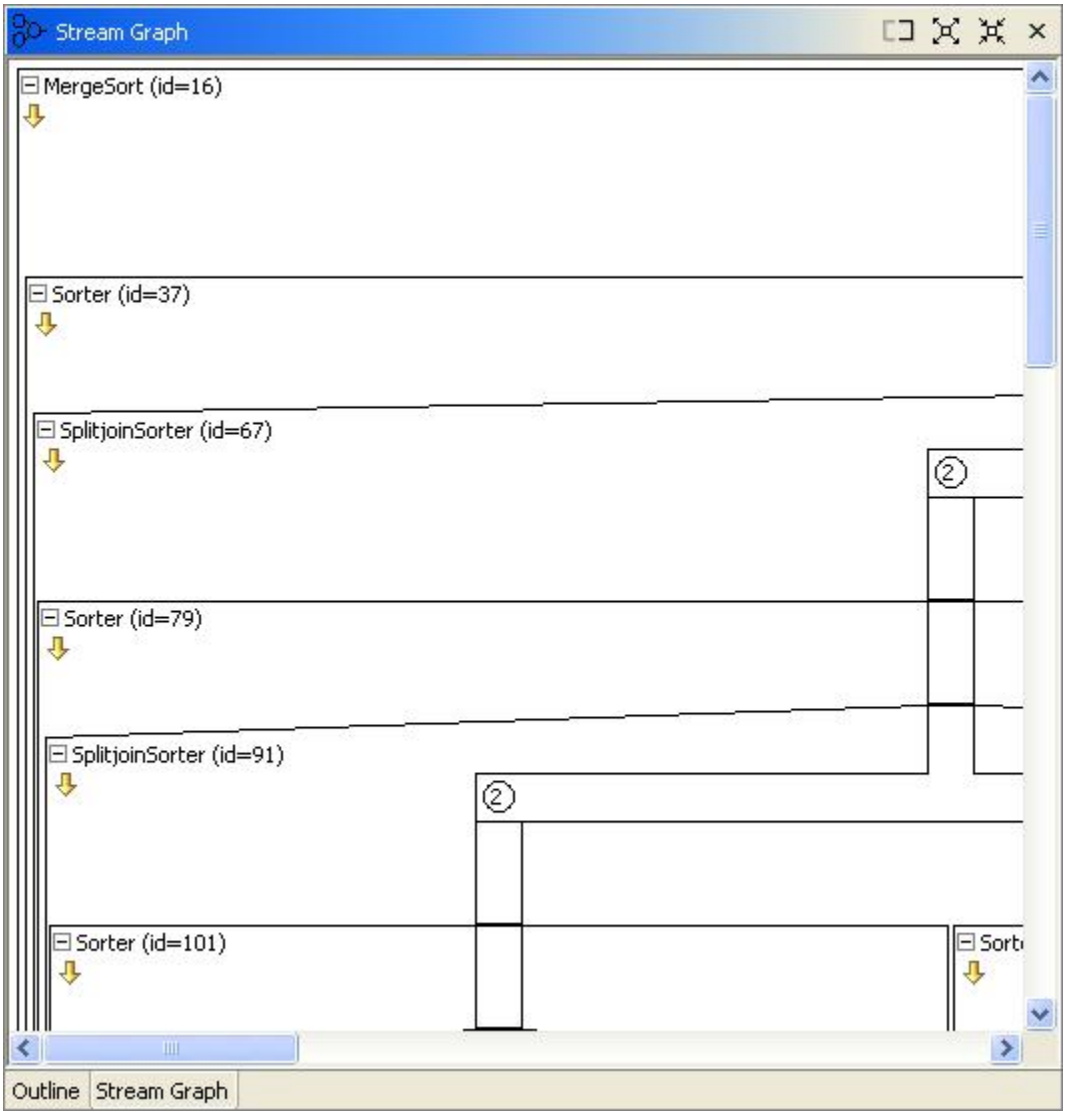


Figure 5-8: A large(r) StreamIt graph.

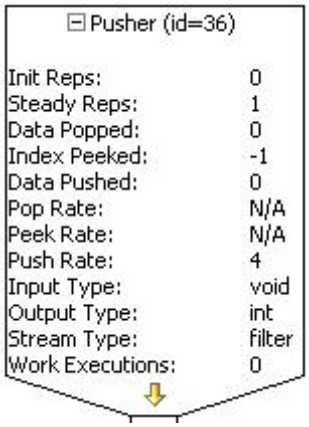


Figure 5-9: An expanded filter with statistics displayed.

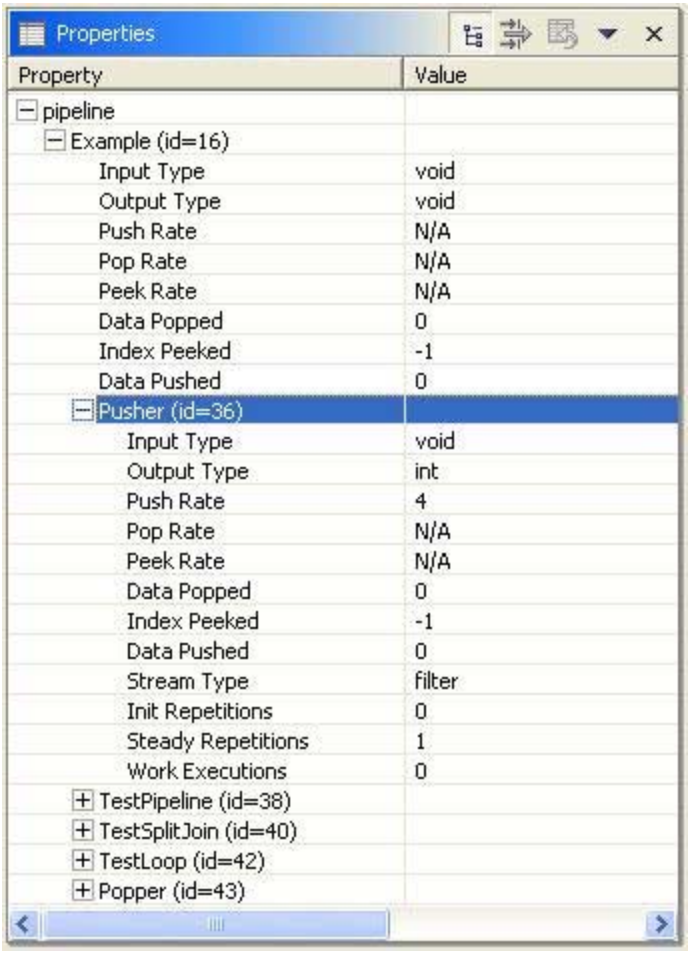


Figure 5-10: The Properties view displaying the statistics for an entire stream graph.

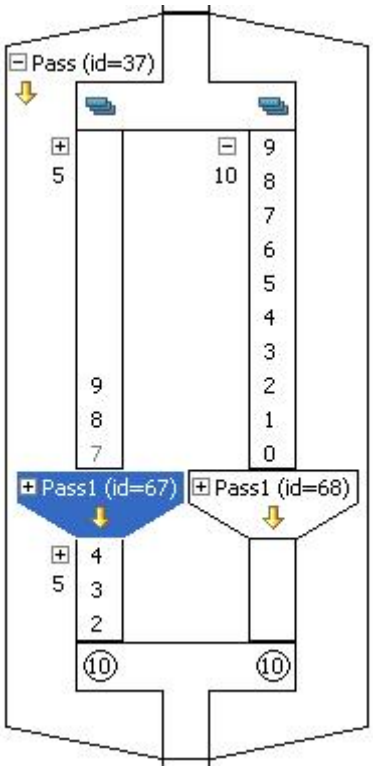


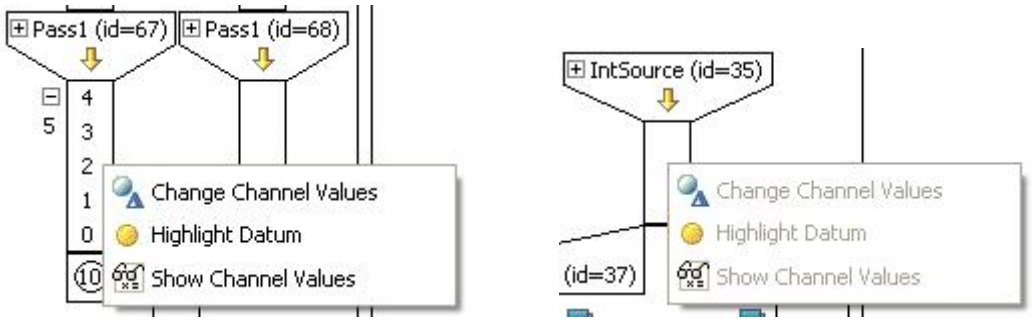
Figure 5-11: The Stream Graph view displays data in channels and grays out peeked at data. Some channels are expanded and some are collapsed.

5.2 Stream Data Display, Modification, and Tracking

During program suspension, the data within channels are drawn in a column, from the source to sink stream (Figure 5-11). By default, the size of all channels in the graph is set to only display three data at a time because the number of data can often grow to un-manageable sizes. Furthermore, data that is within the maximum index peeked in a `work` function is grayed out (data is normally in black text).

The Stream Graph view offers several ways for a programmer to examine data within a stream graph.

- Clicking on a plus icon (Figure 5-7) to the left of a channel expands it lengthwise to reveal all data within it (Figure 5-11).
- Clicking on a minus icon (Figure 5-7) collapses the channel to show only three



(a) (b)

Figure 5-12: The Stream Graph view enables users to show, change, highlight, and unhighlight data (a). These features are disabled when not applicable for a particular channel or datum (b).

data within it (Figure 5-11).

- Pressing the "Collapse All" button (Figure 5-7) in the Stream Graph view's toolbar collapses all channels.
- Pressing the "Expand All" button (Figure 5-7) in the Stream Graph view's toolbar expands all channels.
- Right-clicking on a channel and selecting "Show Channel Values" from the pop-up menu appears under the mouse (Figure 5-12).

The number directly below the plus or minus icon to the left of a channel indicates the number of data within the channel. A channel with three or less data cannot be expanded or collapsed and therefore will not have a plus or minus box next to it. Channels are treated like streams in that their expanded or collapsed state is maintained across program suspensions and the state the user left them in, as described in section 5.1.3.

When a programmer selects "Show Channel Values" from a channel's pop-up menu, a dialog appears displaying all the data within the channel (Figure 5-13). This feature is useful when the width of the data exceeds the width of the channel. Within the dialog, each datum is presented on a new line of a read-only text box, mirroring

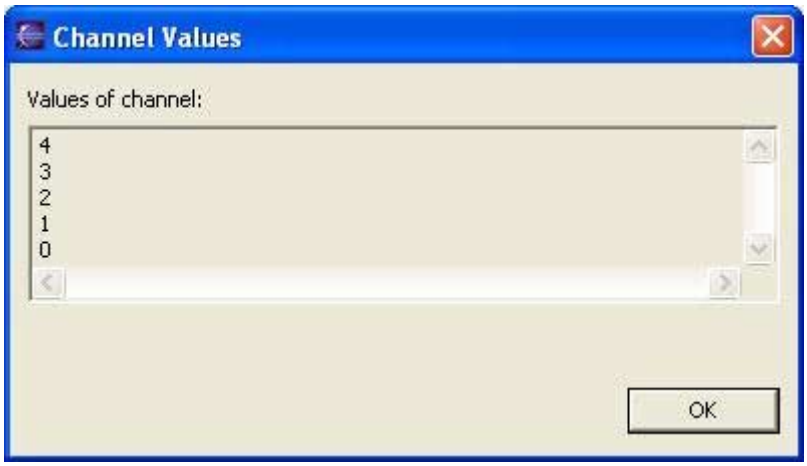


Figure 5-13: A dialog box for showing all of the data within a channel.

the column format of the Stream Graph view. Using a read-only text box prevents the user from trying the change data values, which must be done with the "Change Channel Values" (described below).

Selecting "Change Channel Values" from a channel's pop-up menu (Figure 5-12) lets users change data within channels, much like state variables can be changed in the Variables view. The dialog that appears displays the data within a channel in the same manner as the "Show Channel Values" dialog except that the text box is not read-only (Figure 5-14). Programmers can modify all of the data within a channel at the same time. Entering in invalid data or the wrong amount of data is prevented and causes an "Invalid value(s)" to appear directly below the text box.

While stepping through a work function, data can be tracked with highlighting by right-clicking on an un-highlighted datum and selecting "Highlight Datum" from the pop-up menu that appears or double-clicking on an un-highlighted datum (Figure 5-12). Data can be un-highlighted in an analogous way.

"Show Channel Values," "Change Channel Values," "Highlight Datum," and "Un-highlight Datum" are disabled when no data exists within the channel or at a particular location within the channel.

Figure 5-14: A dialog box for modifying all of the data within a channel.

Chapter 6

Results

A user study was designed and executed to assess the efficacy of the SDT. The goals of the user study were two-fold. The first goal was to discover problems with the SDT and to generate feature suggestions. In addition to generic bugs and glitches within the SDT itself, difficulties with functionality, performance, reliability, and usability (efficiency, subjective satisfaction, and ability to learn, remember, and avoid making errors) were specifically targeted. As seen in section 6.2, addressing these goals was done mainly through questionnaires and programmatic logging of user actions. The second goal was to gather data and qualitative findings that might be used to draw conclusions suggesting that the SDT improved a programmer's ability to debug StreamIt applications. The SDT's impact on improvement was quantified by the time required for a user to successfully debug an application. In addition, the design decisions of the study's setup were influenced by the need to minimize as much as possible threats to the internal validity (ordering and selection effects), external validity (population sampling, training relevancy, and task relevancy), and reliability (user differences and measurement error) of the study.

This chapter discusses the target user population (6.1), procedures used in the study (6.2), a pre-study in-house evaluation (6.3), salient details that occurred during the study (6.4), quantitative results (6.5), qualitative results (6.6), and lessons learned (6.7) from the user study.

6.1 Target Population

In the long term, the target population of the StreamIt language and SDT is streaming application programmers who habitually code in assembly, C, C++, etc. Unfortunately, as a relatively new language [43] still under development, StreamIt is used mainly in research academia at major universities, such as MIT, Stanford [5] [15], and Berkeley [21], with low usage in professional settings. Consequently, the target population of this user study is potential streaming application programmers: computer science students experienced in general-purpose languages such as C, C++, Java, etc. who specialize in fields such as communications, signal processing, computer systems and architecture, and systems, decision, and control. Since these fields require knowledge of more advanced computer science topics, target users are probably graduates and experienced undergraduates. In terms of general tool experience, users probably have intermediate to advanced knowledge of program visualization tools, debuggers, and programming in a UNIX or Windows development environments.

Users were found for this study through solicitations to computer science graduate and undergraduate MIT mailing lists. The nature of the programming language and development tool were not divulged to avoid potential users studying StreamIt or the SDT before the day of the study. Moreover, in order to screen participants for the qualities detailed above, these solicitations asked the potential users for a description of their programming experience and their year in school.

6.2 Procedure

The study's setup was to present each user with an ordered set of documents describing tasks to complete using the SDT:

(1) Pre-Study Questionnaire: This document was designed to gather information on the participant's programming background and skill level. Questions such as year in school, major, degree being sought, area of computer science concentration, relevant classes, language proficiency, application development experience,

and background in DSP, IDE, and the SDT are asked (appendix A.1).

(2) StreamIt Language Tutorial: This written presentation was intended to give a baseline knowledge of the StreamIt language to all users. It describes and illustrates the syntax and semantics of the StreamIt language, covering all of the topics discussed in section 3.1. Furthermore, example toy applications and tips on the most common mistakes new StreamIt programmers are likely to make were included.

(3) SDT Tutorial: Another written presentation, this document was aimed at informing users of the essential features of the SDT. The first part of the tutorial describe and illustrate the functionality of each view and editor in the StreamIt and Debug perspectives (appendix A.2). For the second portion of the document, the tutorial of the help manual described in section 4.5 was adapted by condensing its length and breadth of features. The resulting tutorial contains step-by-step instructions on how to compile, run, and debug a sample application.

(4) User Tasks: This document (appendix A.3) instructs users to debug nine StreamIt applications, each containing one or more bugs (see appendix refsec:appscode for code listings and descriptions). These programs were to be debugged sequentially, some with and some without the SDT. As each application is debugging, users are asked to record their start and end times, the debugging methods they used, and a short diagnosis of each bug.

(5) Description of Applications and Code: This document contains a description of each application (numbered 1 through 9), a code listing, a sample of buggy output, and a sample of correct output (appendix A.4).

(6) Post-Study Questionnaire: This document was designed to gather data on the participant's experience debugging with and without the SDT. Furthermore, questions pertaining to the perceived difficulty of each problem and a general description of how the user debugged each application are asked, in addition to

questions concerning bugs, satisfaction, ability to learn and remember, speed after learning, ease of use, functionality, comments, and improvements related to the SDT (appendix A.5).

(7) Post-Study Interview: Instead of completing a document, users participated in an interview concerning their overall experience with the SDT. General questions about their satisfaction, debugging difficulties, suggested improvements, etc. with the SDT were asked.

Both StreamIt and SDT tutorials were written, rather than oral, presentations because learning by reading informational material is more akin to how programmers normally learn new languages. Furthermore, written material also maintained consistency across the three sessions of the study and across users for reliability.

In this study, debugging without using the SDT is defined as using only Eclipse's default text editor, the SDT's ability to launch applications in run mode, and the Console view. Since switching between debugging with and without the SDT might have been confusing to some users, preventative measures against inappropriately using the SDT were taken by programming the SDT to check a permissions file in the user's workbench.

In order to control against the SDT's effect on a programmer's debugging ability and ensure internal validity, users were grouped into four categories, which were based on which applications were to be debugged with and without the SDT (as seen in Table 6.1). All users were asked to debug application 1 without the SDT and application 2 with the SDT to assess starting ability and create a baseline reference for later comparison. Moreover, the simplicity of their purpose and bugs was aimed at bolstering the user's confidence with easy warm-ups. Then, half of the users (group A) were told to debug applications 3, 4, and 5 with the SDT and 6, 7, and 8 without the SDT. Meanwhile, the other half (group B) were told to debug 3, 4, and 5 without the SDT and 6, 7, and 8 with the SDT to avoid ordering effects. Due to this grouping structure, applications 3 and 6, 4 and 7, and 5 and 8 were written to be of comparable difficulty. For application 9, half of group A (A1) and half of group

Application	Group A		Group B	
	A1	A2	B1	B2
1. BitTwiddle	Without	Without	Without	Without
2. Fib	With	With	With	With
3. EchoEffect	With	With	Without	Without
4. MergeSort	With	With	Without	Without
5. Cornerturn	With	With	Without	Without
6. EchoEffect2	Without	Without	With	With
7. BubbleSort	Without	Without	With	With
8. BitReverse	Without	Without	With	With
9. Overflow	With	Without	With	Without

Table 6.1: User groupings describing whether the SDT was or was not used for each application.

B (B1) were asked to debug with the SDT, while the other halves (A2 and B2) were asked to debug without the SDT. Cross-sectioning the groups was aimed at ensuring external validity.

The entire study was intended to take two hours, with roughly 45 minutes spent on the tutorials and the rest of the time spent on debugging. Participants were asked to complete the set of documents at their own pace. All of the material and software involved in the study was made available on pre-configured, pre-installed computers. Although users were allowed to ask questions throughout the study, answers related solving a particular bug or performing a particular SDT operation were not given. Upon completion of the study, participants received a $40 gift certificate.

6.3 In-House Evaluation

An in-house evaluation was held as a pilot test several days before the study. The goal of the in-house evaluation was to fix confusing material (e.g. questionnaires, instructions, tutorials, tasks, etc), streamline the test procedures, and refine the study's applications. Three designers of the StreamIt language and compiler participated in the dry run of the study, which uncovered several problems with the study (beyond small bugs) as it was then. This section describes the salient issues below.

- Most of the documents had either too little or too much information. The questionnaires and tasks files did not have detailed enough instructions, causing the in-house evaluators to waste time on figuring out what to do, rather than doing it. The applications and code listings did not provide high-level descriptions and the code contained little comments and used complicated variables names. In contrast, the SDT tutorial contained too much depth and breadth of features, as the original idea was to ask users to read the entire tutorial of the help manual.

- The applications themselves were considered much too difficult, especially given that some were taken from real benchmark applications. One evaluator gave up after 15 minutes because he could not figure out what the application was trying to do. He expressed concern that users would complete a few problems and then give up. The evaluators recommended smaller, synthetic examples that test basic knowledge of StreamIt and suggested new applications, which resulted in applications 1, 3, 6, and 8 to replace the more difficult examples. Furthermore, one of the sorting applications had an unrealistic bug, as it lacked any comparison code.

- Debugging the applications also yielded new feature suggestions from the evaluators: The display of roundrobin weights within the Stream Graph view for both splitters and joiners, maintaining a view of the stream graph after a program had run to completion, the ability to debug multiple launches simultaneously, and stepping through the execution of the splitter or joiner. These features were all added to the SDT except for the last suggestion, due to time constraints. This suggestion is a candidate for future work, as discussed in section 7.1.

After making the recommended changes to the study materials, another iteration of the in-house evaluation was done. However, most of the problems uncovered were minor bugs.

6.4 During the Study

The study was divided into three sessions over a three-day period, and a cluster of Windows XP machines was used. Although each session was intended to last around two hours, the sessions on average took four hours for the last user to finish. Many users were either unable or did not have enough time to debug certain applications.

Even though 25 users were scheduled to participate (5 people for sessions 1 and 2 and 15 people for session 3), cancellations reduced the participation to 20 users and led to uneven groupings: There were 6 people in A1, 5 in A2, 4 in B1, and 5 in B2. Of the 20 participants, there were 4 juniors, 2 seniors, 8 masters, and 6 Ph.D. students, all majoring in Electrical Engineering and Computer Science. Although none had used the SDT in the past, 6 users had experience with DSP.

During the study (as well as in the user tasks and post-study questionnaire), the majority of questions asked by participants centered around a core set of issues:

(1) Debugging without the SDT (see section 6.2).

(2) The execution model and the effect of suspending within `init` functions (see section 3.1.7).

(3) The number of iterations of the top-level pipeline in comparison to the number of executions of a filter's `work` function (see section 3.1.7).

(4) The meaning of peek as a maximum index rather than amount of data peeked (see section 3.1.2).

(5) Changing between the Debug and StreamIt perspectives.

(6) Confirming that a user could not use the SDT for application 9.

Although questions 1 through 4 were explained in the tutorials provided, a likely reason for this pattern is that not enough time was budgeted for learning the StreamIt language and tutorial, as indicated by the actual length of the sessions. Issue (5) was a clear bug in the study, as the tutorial did not include information on how

to change between different perspectives. As a result, users wasted time trying to change perspectives rather than debug applications, which impacts the reliability of the study's results. For issue (6), application 9's size and ordering probably frustrated users who were not allowed to use the SDT for application 9, resulting in those users questioning whether their directions were correct.

6.5 Quantitative Results

In general, the quantitative data gathered from the study concerning the SDT's ability to improve a programmer's productivity in debugging StreamIt applications is inconclusive—the SDT helped productivity only in some cases. Attempts to find patterns in the data by normalizing against the control applications (1 and 2) were mostly futile.

Figure 6-1 compares the number of users who solved (and did not solve) each of the applications with (and without) the SDT. Because the groupings are uneven as previously mentioned, the numbers seen in Figure 6-1 are weighted depending on which group is lacking users. The percentage above each quadruple of columns represents the percentage increase or decrease in solving each application caused by using the SDT. On average, 1.56 (32.97%) less users solved applications 3, 4, and 5 when using the SDT compared to participants not using the SDT, while 2.56 (41.76%) more users solved applications 6, 7, 8, and 9 when using the SDT compared to participants not using the SDT.

Figure 6-2 compares the average time spent on each application. The percentage above each pair of columns represents the percentage improvement or deficiency in time caused by using the SDT. On average, users took 7.78 (36.48%) more minutes to solve applications 3, 4, 5, 6, 7, and 9 with the SDT compared to participants not using the SDT, while users took 0.44 (5.19%) less minutes to solve application 8 with the SDT compared to participants not using the SDT. Furthermore, users could not solve a problem spent an average of 16.96 (83.35%) more minutes on applications 6, 7, and 8 with the SDT and 10.67 (-293.9%) less minutes with the SDT for applications

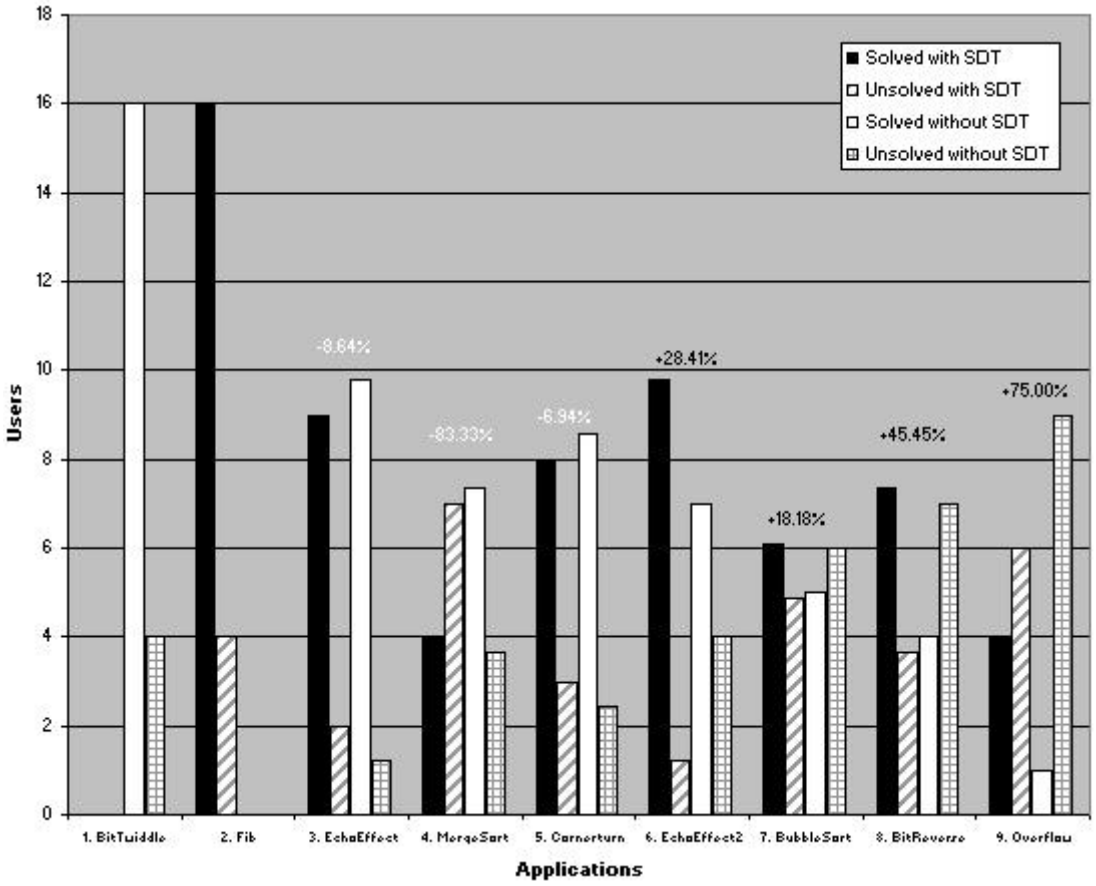


Figure 6-1: Graph comparing the number of users who solved and did not solve each application with and without the SDT.

3, 4, 5, and 9.

A salient trend that emerges from Figures 6-1 and 6-2 is that the SDT may have mitigated user frustration. As previously mentioned, because users generally took much more than the two hours allotted to complete the study, users often became frustrated or rushed with the later applications. Correspondingly, this might cause users to spend less time and solve fewer applications as users progressed through the applications. Although this pattern is true for participants who did not use the SDT, the opposite occurs for participants who used the SDT: Even though 32.97% fewer users were able to solve applications 3, 4, and 5 using the SDT compared to participants not using the SDT, 41.76% more users were able to solve applications 6, 7, 8, and 9 using the SDT compared to participants not using the SDT. Likewise, although users spent -191.93% less time when unable to solve applications 3, 4, and 5 using the SDT compared to participants not using the SDT, users spent 83.35% more time when unable to solve applications 6, 7, and 8. These results suggest that users are willing to spend more time and work on more problems when using the SDT.

Figure 6-3 compares the average normalized time it took user to solve each application with and without the SDT. Since 6 users were not able to solve both application 1 and 2, their times were excluded from this chart. The percentage above each pair of columns represents the percentage improvement or deficiency caused by using the SDT. For applications 3, 6, 8, and 9, the SDT increased debugging time by an average of 241.02% and a median of 37.02%, while the SDT decreased debugging time by an average of 33.4% and a median of 29.0% for applications 4, 5, and 7.

Therefore, although more people were able to successfully debug more applications using the SDT than not using the SDT, the SDT also increases the amount of time it takes to debug some of these applications. However, the SDT is able to decrease the amount of time it takes to debug some applications solved by more users without the SDT. Moreover, if an application's difficulty is measured by the amount of time to solve it, then there seems to be no apparent correspondence between using and not using the SDT and an application's difficulty.

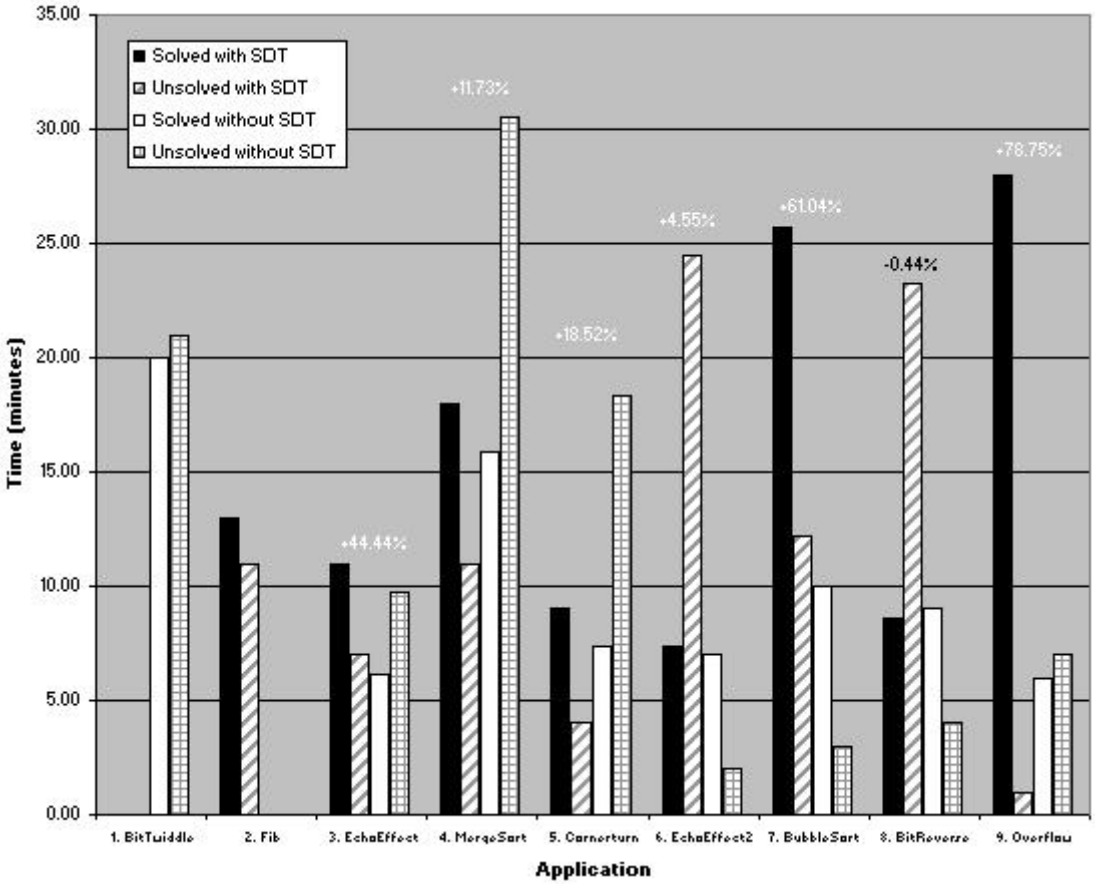


Figure 6-2: Graph comparing the average time it took users to solve or not solve each application with and without the SDT.

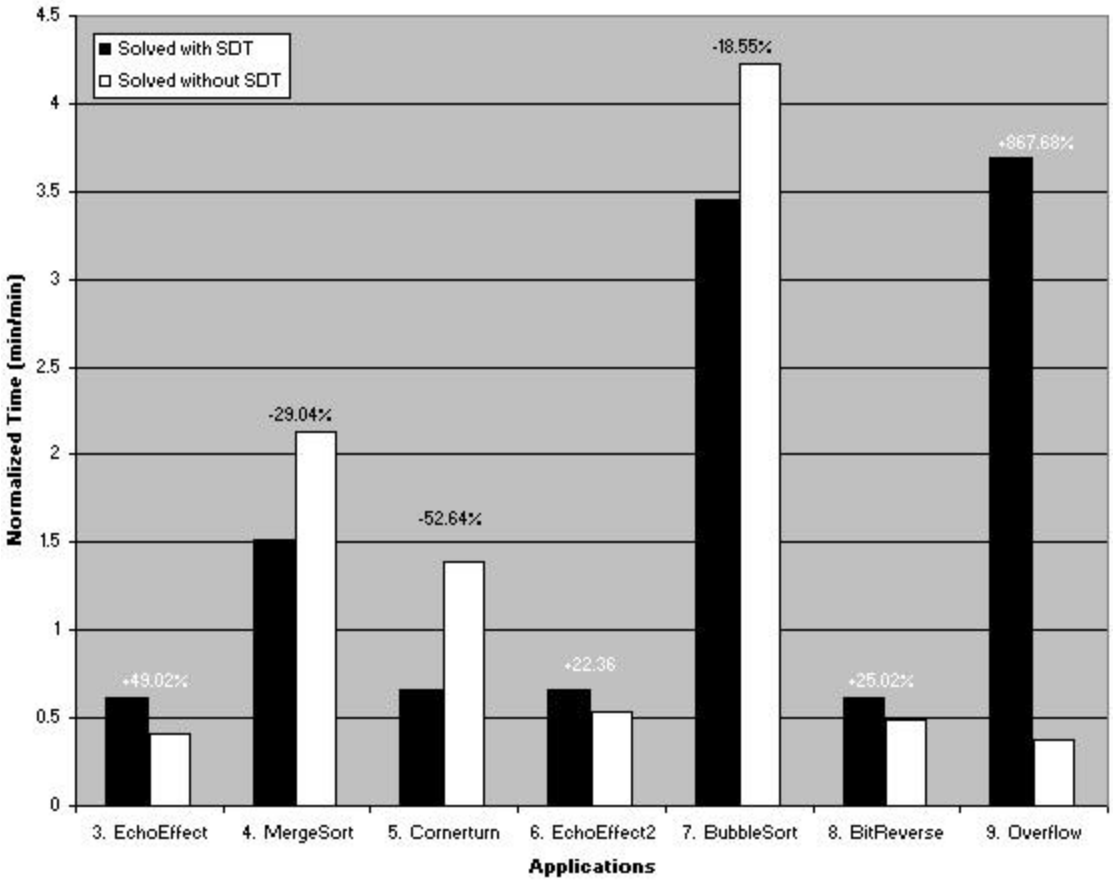


Figure 6-3: Graph comparing the average normalized time it took users to solve applications 3 through 9 with and without the SDT.

6.6 Qualitative Results

In contrast to the previous section, qualitative data obtained from the post-study questionnaire was more productive toward finding new feature ideas and problems with functionality, performance, reliability, and usability.

6.6.1 Problems with the SDT

A major problem that many users found was that the SDT does not display large stream graphs well and does not allow users to view the entire stream graph easily. Specifically, the SDT rendering stream graphs were considered too large and intimidating in comparison to the amount of space available in the Eclipse workbench, as even participants who used Eclipse's functionality for expanding a view to encompass the entire workbench still could not see the entire graph without scrolling. Furthermore, when applications contained streams with many descendants, the stream graph became confusing due to the narrow spacing of the boundaries between parent and child streams. As a result, many users expressed difficulty in distinguishing the various hierarchical levels of large graphs (see Figure 6-4 for an example).

A few users found that using the Overview of Stream Graph view to navigate the Stream Graph view was inconvenient and confusing. On one user's computer, the rectangle in the Overview of Stream Graph view was the same color as the stream graph, which made distinguishing it from complicated stream graphs impossible.

Using breakpoints was another major problem because users spent too much time trying to figure out where to add breakpoints so that they could view the stream graph. As previously mentioned, users would add breakpoints within `init` functions and expect to view the entire stream graph even though at that point it was not fully built. Within `work` functions, users had difficultly stepping through the code, as users would try other stepping functionality (only the step over button is supported by the SDT) or try to step beyond the end of a `work` function (only stepping within a `work` function is supported).

Although the post-study questionnaire was focused on gathering problems with

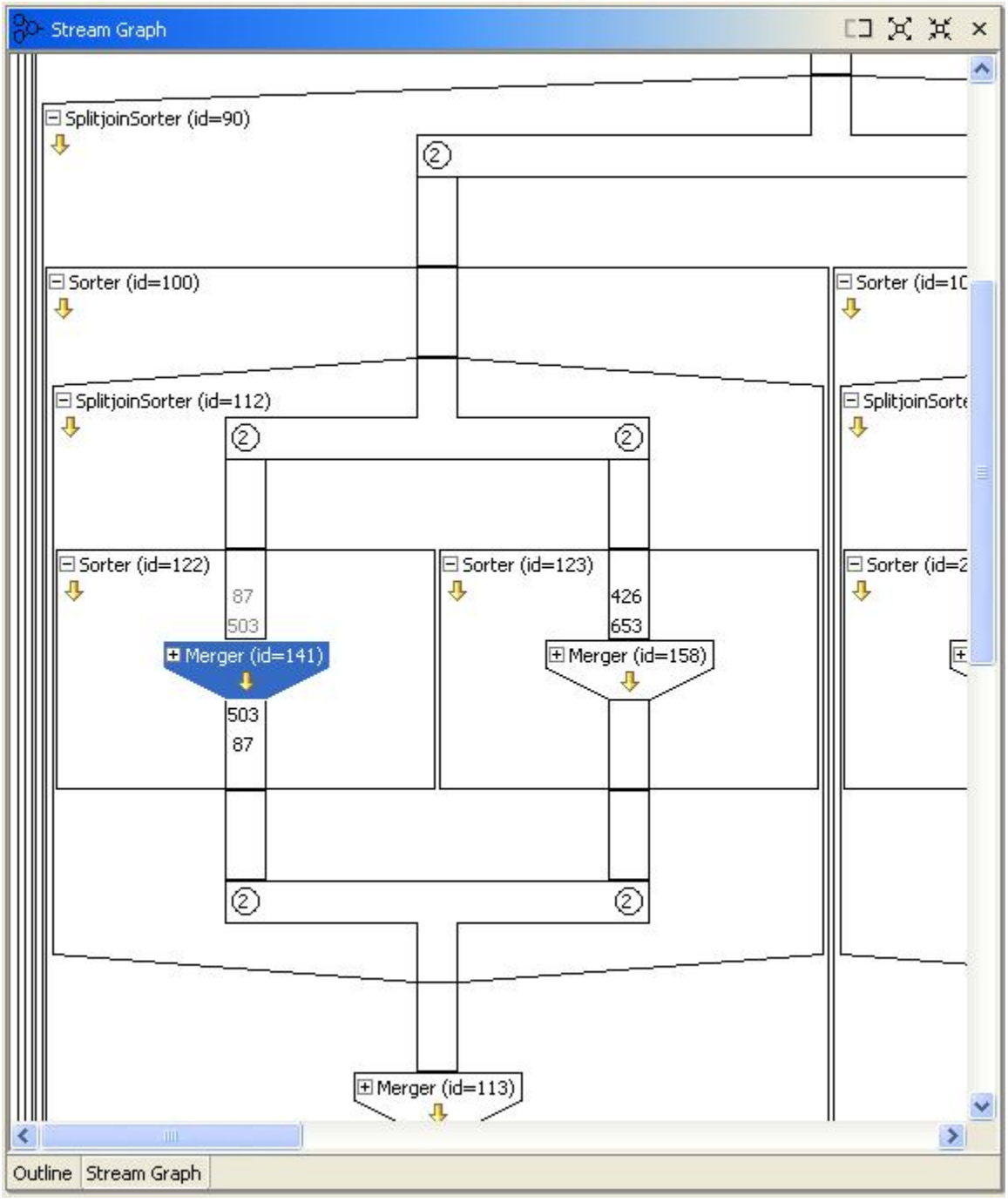


Figure 6-4: The Stream Graph view displaying a portion of Application 4's stream graph.

the SDT, several users commented on their difficulty using Eclipse and the StreamIt language and how they interfered with debugging stream applications. For Eclipse, 10 participants commented that, in general, Eclipse is difficult to get used to and that, specifically, switching between perspectives was confusing and non-intuitive. For the StreamIt language, 5 participants mentioned that they were uncomfortable or unaccustomed to thinking in terms filters and streams, while an equal number also found it difficult or overwhelming to remember concepts such as roundrobin, enqueue, and general syntax.

6.6.2 Usability of the SDT

In general, users found the SDT somewhat easy to use: From a scale of 1 to 10 (1 being very easy and 10 being very hard), participants on average gave the SDT a 3.85. Many cited various aspects of the Stream Graph view, such as visualizing the stream graph easily, expanding and collapsing graph, simultaneously being able to view the code and graph, and viewing data flow. Furthermore, 14 users said they would use the SDT for debugging StreamIt applications, while 5 said they would not and felt more comfortable debugging "by hand"—just with pen and paper or looking at the code (one user would not use StreamIt). Those who said they would use the SDT stressed that it was most useful for graphically seeing what was going on in a StreamIt program, such as program flow, the entire stream graph, data values, etc. Furthermore, this visualization is more important when code inspection or stepping through code is inappropriate or impractical, as in the case of large-scale applications.

Users found it somewhat easy to learn how to use the SDT: From a scale of 1 to 10 (1 being very easy to learn and 10 being very hard to learn), participants on average gave the SDT a 3.55. Many cited the tutorial as a good reference and easy to follow and said that the GUI layout was either intuitive for a beginner to find basic functionality or similar to other IDE that they have used. In general, when a user did not have experience using IDEs, he said that the amount of information, in the form of views, menus, and buttons, was overwhelming to a beginner. Several users commented that there were too many windows and options from which to choose,

while one person said that the Stream Graph view contained so much information that finding vital information immediately was slow.

Users said it was fairly easy to remember how to use the SDT: From a scale of 1 to 10, participants on average gave the SDT a 2.95. Many users wrote that the SDT only had a few concepts to remember. Participants also stated that GUI had simple repeatable actions and was intuitive enough that remembering was not difficult.

Users remarked that the SDT was marginally fast to use: From a scale of 1 to 10 (1 being very slow and 10 being very fast), participants on average gave the SDT a 5.75. Several users observed that the SDT was fast, especially in comparison to other programming environments, because (1) information displayed was easy to find and viewable simultaneously with other information and (2) breakpoints enabled users to narrow in on relevant code. However, for large-scale stream graphs, many users thought that the SDT was too slow in drawing complex structures. Furthermore, when navigating through the stream graph or stepping through the code, the SDT's redrawing capability was frustratingly slow for many users.

6.6.3 Improvements and New Functionality

Users suggested new features and improvements directly related to problems they found with the SDT in section 6.6.1. For the stream graphs, participants suggested a button to expand all streams except filters, clearer markings of input and output channels, being able to view the entire stream graph with using a breakpoint, a history of data that had passed through channels and filters, and customized expansion and collapse of stream graphs that can be saved for a given applications. Although most users wanted improved drawing of large-scale graphs and better zooming, none mentioned specific functionality for this purpose. Another common request was for better code-based debugging support, such as putting breakpoints on pipelines, split-joins, and feedback-loops, better ability to print debug statements, the capacity to step backwards through code, and the ability to step through code across work functions.

6.7 Lessons Learned

Many problems and issues arose in running the study itself. One of the major problems was the time allotted for users to complete the study. As previously mentioned, on average it took the slowest user double the budgeted amount of time. This lack of time negatively impacted users in several ways, all of which contributed to incomplete or unreliable data: users became frustrated and overwhelmed by the amount of information presented to them, users were unable to complete the study due to time constraints, users did not properly fill out the post-study questionnaire, rushed users put down bogus answers, etc.

Better participant screening for DSP-related and general programming experience should also have been done. In the post-study questionnaire, several users commented that they were unaccustomed to thinking in terms of streaming applications, while some users made remarks that suggested they never truly understood how streaming applications work. In general, the more inexperienced users either focused their attentions on Eclipse-related and StreamIt language problems, indicating a lack of IDE and language-learning experience, or relied almost exclusively on breakpoint functionality and print statements. Because several of the applications were designed to test the effectiveness of the Stream Graph view, these applications were purposely written to be too large and complicated for these traditional debugging techniques.

Having a graduated pay scale for compensation may have alleviated some of the above problems, as it would allow the participants to judge whether they could or were willing to complete all of the applications in the study. Nonetheless, the variation in each participant's experience and skill within a user study is a well-documented issue. Usability studies have found that the best users are often 10 times better than the worst users and the fastest quartile of users are twice as fast as the slowest quartile of users [13] [17]. However, because increasing the number of users within a study only narrows the standard deviation of the mean by the square root of the number of users [13], improving reliability of results is an expensive and time-consuming task. For example, in order to double accuracy, the number of participants in this study

would have to be quadrupled to 80 users, which would cost an additional $2400 and 24 man-hours.

Chapter 7

Conclusion

In this chapter, future work is first described, followed by a summary of this thesis.

7.1 Future Work

Future work on the SDT can naturally progress from the improvements, problems, and new functionality discussed throughout chapter 6. Perhaps the most important are improvements aimed at handling large stream graphs. In addition to displaying the entire stream graph, the Stream Graph view could be altered to only show the graph below a certain node specified by the user—in effect "hanging" the hierarchical tree by a non-root node. This change, in combination with allowing a user to specify whether the Stream Graph view would respond to changes in the Debug view, could focus a user's attention on a specific part of the graph that may be causing an error.

Stream graph navigation might be enhanced by providing a way for users to expand all non-filter streams and making the SDT remember the collapsed and expanded state of stream graph across debugging sessions. For data display, enabling users to specific channel width would make viewing larger, multi-digit data easier, while retaining a record of data that had passed through channels and streams might improve I/O-related debugging. Moreover, although feasible, support for tracking—the relationship between data popped and pushed by a filter—has yet to be developed or investigated. Possible functionality might include (1) data coloring depending on

the particular filter(s) through which they are produced or modified and (2) program suspension conditional on data of a certain color passing through another filter.

Breakpoint stepping can also be improved by allowing users to step through code across `work` functions, to view the stepping through a splitter or joiner's execution, and to automatically suspend a program on every `push`, `pop`, or `peek` call within one or more `work` functions. Furthermore, unsupported stepping functionality, such as step into, step return, and step with filters, should also be disabled. Filter instance breakpoints can be expanded to enable manipulation across debugging sessions. Moreover, these breakpoints could be allowed for non-filter streams, which would suspend a program on the `work` function of the first filter fired within the stream.

7.2 Summary

This thesis presents the StreamIt Development Tool, an IDE designed to improve the coding, debugging, and visualization of streaming applications by exploiting the StreamIt language's ability to intuitively represent these applications as structured, hierarchical graphs. Although industry and academia have devoted much effort to tools for developing and debugging software, the SDT aims to emulate the best of traditional debuggers and IDEs while moving toward hierarchical visualization and debugging concepts specialized for streaming applications. As such, utilities for stream graph examination and navigation and stream data display, modification, and tracking are provided, in addition to program creation and code editing, breakpoints, program compilation and launch support, and general debugging and help support.

A user study evaluating the SDT uncovered several problems and areas of improvement that need to be addressed before this tool can approach its goals. Assessment of the SDT's efficacy in its current state is inconclusive—the SDT demonstrates both the ability to improve and hinder a user's debugging ability. Facilitating effective coding and debugging techniques and developing for scalability are critical elements in improving the SDT's effectiveness.

Appendix A

User Study Documents

This appendix contains documents pertaining to the user study.

A.1 Pre-Study Questionnaire

Please complete the below questionnaire.

- Check or uncheck a box ☐ by clicking on it.
- Select from a drop-down box 1 (novice) by clicking on it and selecting from the drop-down list box that appears. The default selection is "1 (novice)."
- Type into a text field _____ by clicking on the field and typing. The text field expands as you type. You can type multiple lines of text by pressing enter.

When you are finished, please save the file as

C:\Program Files\eclipse\workspace\Pre-Study Questionnaire.**doc**

by going to the **File** menu and clicking **Save As...** (notice that the file currently ends in **dot**).

1a. Please check one:

☐ Freshman ☐ Sophomore ☐ Junior ☐ Senior ☐ Graduate

2a. Are you course 6? ☐ Yes ☐ No

2b. If yes, please check: ☐ 6-1 ☐ 6-2 ☐ 6-3 ☐ M.Eng. ☐ Ph.D.

2c. If you are a graduate student in course 6, please check your area(s) of concentration:

☐ Communications
☐ Systems, Decision and Control
☐ Signal Processing
☐ Bioelectrical Engineering
☐ Circuit Design
☐ Devices and Materials
☐ Electromagnetic Energy, Fields, and Waves
☐ Theoretical Computer Science
☐ Computer Systems and Architecture
☐ Artificial Intelligence
☐ Computer Networks
☐ Computer Graphics

3. Please check any of the following courses that you may have taken:

☐ 6.001 ☐ 6.011 ☐ 6.035 ☐ 6.050J ☐ 6.115 ☐ 6.170 ☐ 6.291

☐ 6.341 ☐ 6.821 ☐ 6.893 (UI Design & Implementation) ☐ HST.582J

4. Please list the languages you have programmed in, the number of years you have programmed in each language, and your proficiency in each language. If you have ever used Matlab or programmed in StreamIt, please include it in the table below.

Language	Years	Proficiency (1 is novice, 10 is expert)
		1 (novice)
		1 (novice)
		1 (novice)
		1 (novice)
		1 (novice)
		1 (novice)

5. For the 4 largest programs that you have written, please list the language you wrote each in, a short description of each program, and the approximate lines of codes for each program.

Problem	Language	Description	Approximate Lines of Code
1			
2			
3			
4			

6a. Do you have any background in digital signal processing (DSP)? ☐ Yes ☐ No

6b. If yes, please describe the nature of this background (i.e. work experience, research, etc):

7a. Have you ever used an IDE (e.g. Visual Studio, Eclipse) before today?

☐ Yes ☐ No

7b. If yes, please list the IDEs you have used, the number of years you have used them, and your proficiency in using them.

IDE	Years	Proficiency (1 is novice, 10 is expert)
		1 (novice)
		1 (novice)
		1 (novice)
		1 (novice)
		1 (novice)
		1 (novice)

8a. Have you ever used the StreamIt Development Tool (SDT) before today?

☐ Yes ☐ No

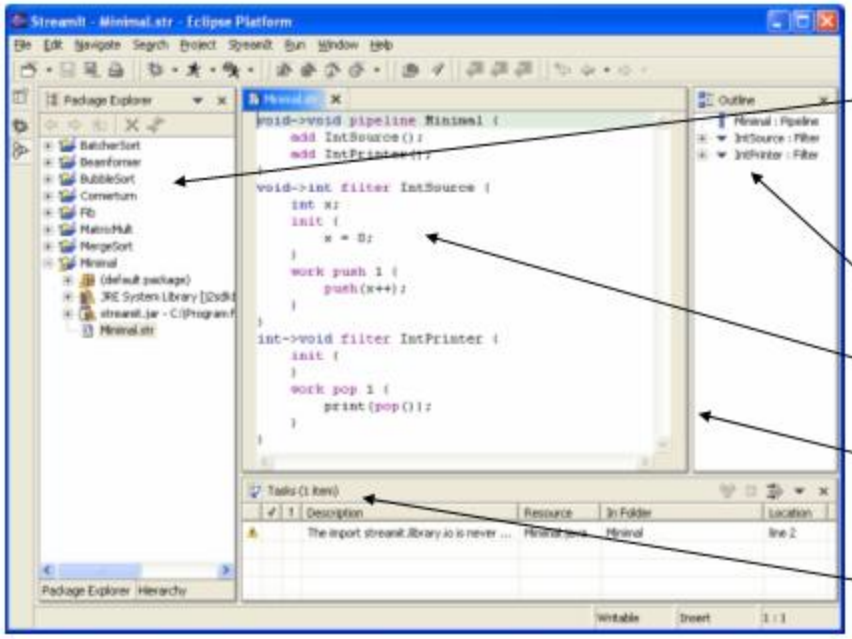
8b. If yes, how many month have you used the SDT and how proficient you are in using the SDT?

Months: Proficiency: 1 (novice)

A.2 SDT Tutorial

The StreamIt Perspective

Below is a screenshot of Eclipse's desktop environment, called the *workbench*, when it is in the StreamIt perspective. The StreamIt perspective defines a group of windows, called *views*, related to editing StreamIt files.



The Package Explorer View displays the projects available to you and allows you to open StreamIt files by double-clicking them. Each StreamIt file is associated with a project of the same name. You can reveal the contents of a project by expanding the "plus" icon next to it.

The Outline View displays an outline of the currently opened StreamIt file.

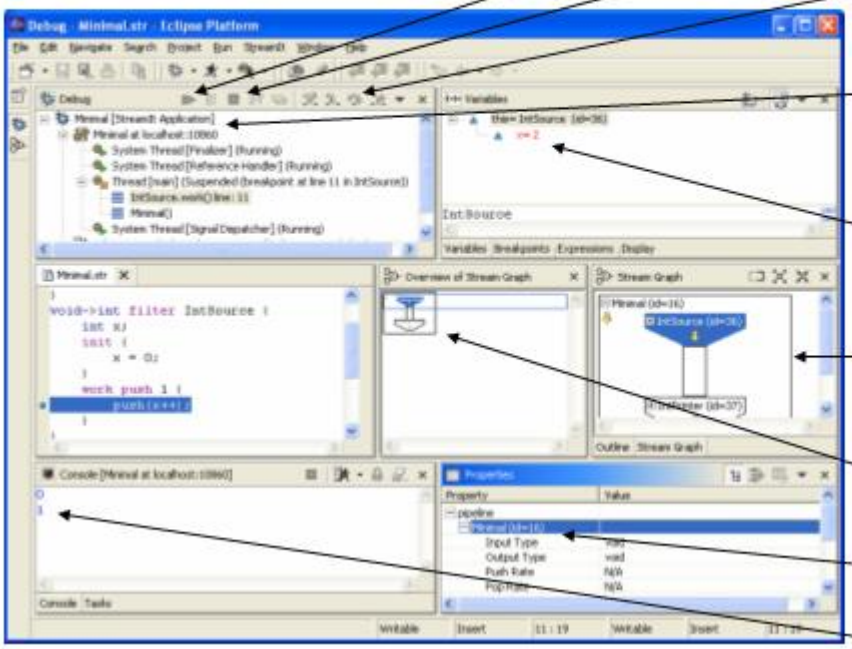
You can edit StreamIt files in this area.

To resize any of the views or editors, put your cursor above a window border and drag.

The Tasks View displays information about the currently opened StreamIt file, such as warnings or errors.

The Debug Perspective

Below is a screenshot of the workbench in the Debug perspective. The Debug perspective defines the views related to running StreamIt files.



Resume program.

Terminate program.

Step to next line of executable code.

The Debug view displays the processes, threads, and stack frames of the StreamIt program being debugged.

The Variables view displays program variables. Only the variables in the stackframe selected in the Debug view are visible.

The Stream Graph view displays the stream graph of the StreamIt program currently being debugged.

The Overview of Stream Graph view displays a zoomed-out version of the Stream Graph view.

The Properties view displays program metrics.

The Console view displays program output.

A.3 User Tasks

Table 1 below lists nine StreamIt applications you will find in your Eclipse workspace. Each application contains one or more bugs. A description of each application, its currently incorrect output, its code, and correct sample output are provided in the next handout.

Please debug these applications in number order (1-9) and one at a time. We would like you to debug some of these applications using the SDT and some without* using the SDT. The third column of Table 1 indicates whether you should use the SDT for each application. Please save any changes you make to the code in your workspace.

In Table 1's fourth and fifth columns, please record your start and end times (in hours and minutes) for each application. Please leave these columns blank if you are unable to start or finish any application.

- Check or uncheck a box ☐ by clicking on it.
- Type into a text field by clicking on the field and typing. The text field expands as you type. You can type multiple lines of text by pressing enter.

After you debug each application, please fill out the information in Table 2 (found after Table 1). The second column of Table 2 is for a one sentence diagnosis of what is wrong with each application. The third column of Table 2 is for checking any of the debugging methods that you used for each application.

It is very important for the sake of this study that you attempt all of these applications. We expect that you will probably spend a little over an hour on all of these problems. While debugging a certain application, if you find that you may not be able to finish the rest of the applications, move on to the next one. If you have extra time at the end, you can return to any unfinished applications.

When you are finished, please save the file as

C:\Program Files\eclipse\workspace\User Tasks.**doc**

by going to the **File** menu and clicking **Save As ...** (notice that the file currently ends in **dot**).

*Please use Eclipse's Text Editor and Run functionality: StreamIt files can be opened with the Text Editor by right-clicking on the file in the Package Explorer view and selecting **Open With > Text Editor**. StreamIt applications can be run by clicking on **Run > Run As > StreamIt Application**. Please do not use **Debug History**, **Debug As**, or **Debug ...** for the applications to be debugged without the SDT.

Table 1

Application	Project Folder	Debug with SDT	Start Time (hr:min)	End Time (hr:min)
1	BitTwiddle	No	:	:
2	Fib	Yes	:	:
3	EchoEffect		:	:
4	MergeSort		:	:
5	Cornerturn		:	:
6	EchoEffect2		:	:
7	BubbleSort		:	:
8	BitReverse		:	:
9	Overflow		:	:

Table 2

Application	One-Sentence Bug Diagnosis	Debugging Method (check all that apply)
1		☐ code inspect ☐ print statements ☐ SDT
2		☐ code inspect ☐ print statements ☐ SDT
3		☐ code inspect ☐ print statements ☐ SDT
4		☐ code inspect ☐ print statements ☐ SDT
5		☐ code inspect ☐ print statements ☐ SDT
6		☐ code inspect ☐ print statements ☐ SDT
7		☐ code inspect ☐ print statements ☐ SDT
8		☐ code inspect ☐ print statements ☐ SDT
9		☐ code inspect ☐ print statements ☐ SDT

A.4 Description of Applications and Code

A.4.1 Application 1: BitTwiddle

The purpose of the code on the next page is to drop every third bit of a 96 bit stream.

The correct output for 1 iteration is:

```
0
1
3
4
6
7
9
0
2
3
5
6
8
9
1
2
4
5
7
8
```

Instead, the code is producing the following output:

```
0
1
3
4
5
7
8
9
1
2
3
5
6
7
9
0
1
3
4
5
7
8
9
1
2
3
5
6
7
9
```

```
void->int filter Source(int N, int
    drop, int set) {
    int x;
    init {
        x = 0;
    }
    work push set {
        for (int i = 0; i < set; i++) {
            push(x++);
            if (x == 10 || x == N*drop)
                x = 0;
        }
    }
}

int->void filter IntPrinter {
    init {
    }
    work pop 1 {
        print(pop());
    }
}

int->int filter DropBit(int drop, int
    set) {
    init {
    }
    work pop set push drop {
        for (int i = 1; i <= set; i++){
            if (i == drop) {
                pop();
            } else {
                push(pop());
            }
        }
    }
}

void->void pipeline BitTwiddle {
    // set is the amount of data that
    // DropBit will consider
    int N = 32;
    int drop = 3;
    int set = 4;
    add Source(N, drop, set);
    add DropBit(drop, set);
    add IntPrinter();
}
```

A.4.2 Application 2: Fibonacci

The purpose of this code is to generate Fibonacci numbers, which form a sequence defined recursively by:

```
f(0) = 0
f(1) = 1
f(n) = f(n - 2) + f(n - 1)
```

The correct output for 10 iterations is:

```
1
2
3
5
8
13
21
34
55
89
```

Instead, the code is producing the following output:

```
1
2
2
3
4
5
7
9
12
16
```

```
void->void pipeline Fib {
    add FibLoop();
    add IntPrinter();
}

void->int feedbackloop FibLoop {
    join roundrobin(0, 1);
    body PeekAdd();
    loop Pass();
    split duplicate;
    enqueue 0;
    enqueue 1;
    enqueue 1;
}

int->int filter PeekAdd {
    work push 1 pop 1 peek 2 {
        int larger = peek(1);
        int smaller = pop();
        int result = smaller + larger;
        push(result);
    }
}

int->void filter IntPrinter {
    work pop 1 {
        print(pop());
    }
}

int->int filter Pass {
    work push 1 pop 1 peek 1 {
        push(pop());
    }
}
```

A.4.3 Application 3: EchoEffect

The purpose of this code is to simulate how echos are added to sound waves, which can be represented as digital data. Echos are added to digital data by adding the original sequence of data to a time-shifted version of the original sequence. In the case of EchoEffect, the code is trying to add data at t to the data at $t + 1$. For the sake of simplicity, the input is a sequence of consecutive numbers "0, 1, 2, 3, ..." where the first value is at $t = 0$.

The correct output for 10 iterations is:

```
1
3
5
7
9
11
13
15
17
19
```

Instead, the code is producing the following output:

```
2
4
6
8
10
12
14
16
18
20
```

```
void->int filter IntSource {
    int x;
    init {
        x = 0;
    }
    work push 1 {
        push(x++);
    }
}

int->void filter IntPrinter {
    init {
    }
    work pop 1 {
        print(pop());
    }
}

int->int filter Original {
    init {
    }
    work push 1 pop 1 {
        int x = pop();
        push(x);
    }
}

int->int filter Delayed {
    init {
    }
    work push 1 pop 1 peek 3 {
        pop();
        push(peek(1));
    }
}

int->int splitjoin Echo() {
    split duplicate;
    add Original();
    add Delayed();
    join roundrobin;
}

int->int filter Adder {
    init {
    }
    work push 1 pop 2 {
        int x = pop();
        int y = pop();
        push(x + y);
    }
}

void->void pipeline EchoEffect() {
    add IntSource();
    add Echo();
    add Adder();
    add IntPrinter();
}
```

A.4.4 Application 4: MergeSort

The purpose of the code on the next page is to implement a mergesort algorithm, sorting a sequence of 16 numbers recursively. In general, mergesort operates by decomposing the input sequence into two halves. Each half is sorted by involving the mergesort algorithm recursively. Then, the two sorted halves are merged into a sorted sequence.

The correct output for 1 iteration is:

```
61
87
154
170
275
426
503
509
512
612
653
677
703
765
897
908
```

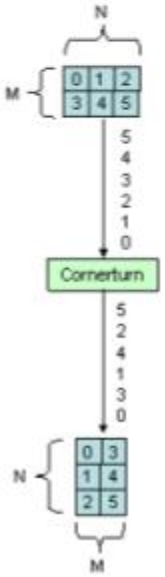
Instead, the code is producing the following output:

```
87
61
170
154
426
275
503
512
612
653
509
703
677
765
908
897
```

```
void->void pipeline MergeSort {
    int SIZE = 16;
    add IntSource(SIZE);
    add Sorter(SIZE);
    add IntPrinter();
}

// An input of SIZE random numbers to
sort.
void->int filter IntSource(int SIZE) {
    int [SIZE]data;
    int index = 0;
    init {
        data[0] = 503; data[1] = 087;
        data[2] = 512; data[3] = 061;
        data[4] = 908; data[5] = 170;
        data[6] = 897; data[7] = 275;
        data[8] = 653; data[9] = 426;
        data[10] = 154; data[11] = 509;
        data[12] = 612; data[13] = 677;
        data[14] = 765; data[15] = 703;
    }
    work push 1 {
        push(data[index++]);
        if (index == SIZE) {
            index = 0;
        }
    }
}

/**
 * The merger component of the merge
 * Combines two sorted streams into
 * another sorted stream, producing a
 * total of <N> elements.
 */
int->int filter Merger (int N) {
    init {
    }
    work push N pop N {
        // initialize indices
        int index1 = 0;
        int index2 = 1;

        // merge values
        while (index1<N && index2<N) {
            int val1 = peek(index1);
            int val2 = peek(index2);
            if (val1 <= val2) {
                push(val1);
                index1 += 2;
            } else {
                push(val2);
                index2 += 2;
            }
        }

        // merge remainder if one
        // stream dries out
        int leftover = index1 < N ?
                       index1 : index2;
        for (int i=leftover; i<N;i+=2){
            push(peek(i));
        }

        // pop all the inputs
        for (int i=0; i<N; i++) {
            pop();
        }
    }
}

int->int splitjoin SplitjoinSorter(int
    N) {
    split roundrobin(2, 2);
    add Sorter(N/2);
    add Sorter(N/2);
    join roundrobin(2, 2);
}

// Sorts a stream of integers.
int->int pipeline Sorter (int N) {
    // if we have more than two items,
    // then sort in parallel
    if (N>2) {
        add SplitjoinSorter(N);
        add Merger(N);
    } else {
        add Merger(2);
    }
}

int->void filter IntPrinter() {
    init {
    }
    work pop 1 {
        print(pop());
    }
}
```

A.4.5 Application 5: Cornerturn

The purpose of the code on the next page is to implement a *cornerturn*, which is essentially a transpose algorithm (exchanging a matrix's rows and columns so that M-row by N-column matrix would become an N-row by M-column matrix). The cornerturn algorithm takes as input a stream whose first N elements are the first row of the input matrix, whose N + 1 to 2N elements are the second row of the input matrix, …, whose N*(M – 1) + 1 to N*M elements are the Mth row of the input matrix. The cornerturn algorithm should output a stream whose first M elements are the first column of the input matrix, whose M + 1 to 2M elements are the second column of the input matrix, … whose M*(N – 1) + 1 to M*N elements are the Nth column of the input matrix. Visually, this can be expressed as:



The correct output for 1 iteration is:

```
0.0
3.0
1.0
4.0
2.0
5.0
```

Instead, the code is producing the following output:

```
0.0
2.0
4.0
1.0
3.0
5.0
```

```
void->void pipeline Cornerturn {
    int M = 2;
    int N = 3;
    add Source(M, N);
    add Transpose(M, N);
    add Sink(M, N);
}

float->float filter Pass {
    work push 1 peek 1 pop 1 {
        push(pop());
    }
}

void->float filter Source(int M, int N)
{
    int x;
    init {
        x = 0;
    }
    work push M*N {
        for (int i = 0; i < M*N; i++) {
            push(x++);
        }
    }
}

float->float splitjoin Transpose(int M,
    int N) {
    split roundrobin();
    for (int i = 0; i < M; i++) {
        add Pass();
    }
    join roundrobin(N);
}

float->void filter Sink(int M, int N) {
    work peek M*N pop M*N {
        for (int i = 0; i < M*N; i++) {
            print(pop());
        }
    }
}
```

A.4.6 Application 6: EchoEffect2

This code is also trying to add an echo to digital data. However, in this case the code is trying to add data at t to the data at $t - 1$. For the sake of simplicity, the input is a sequence of consecutive numbers "0, 1, 2, 3, ..." where the first value is at $t = 0$ and the value is assumed to be 0 for $t < 0$.

The correct output for 10 iterations is:

```
0
1
3
5
7
9
11
13
15
17
```

Instead, the code is producing the following output:

```
0
1
2
4
6
8
10
12
14
16
```

```
void->int filter IntSource {
    int x;
    init {
        x = 0;
    }
    work push 1 {
        push(x++);
    }
}

int->void filter IntPrinter {
    init {
    }
    work pop 1 {
        print(pop());
    }
}

int->int filter Original {
    init {
    }
    work push 1 pop 1 {
        push(pop());
    }
}

int->int filter Delayed {
    init {
    }
    work push 1 pop 2 {
        push(pop());
        pop();
    }
}

int->int feedbackloop Echo() {
    join roundrobin(1, 1);
    body Original();
    loop Delayed();
    split duplicate;

    enqueue(0);
    enqueue(0);
}

int->int filter Adder {
    init {
    }
    work push 1 pop 2 {
        int a = pop();
        int b = pop();
        push(a + b);
    }
}

void->void pipeline EchoEffect
    add IntSource();
    add Echo();
    add Adder();
    add IntPrinter();
}
```

A.4.7 Application 7: BubbleSort

The purpose of the code on the next page is to implement a bubblesort algorithm, sorting a sequence of 16 numbers iteratively. In general, bubblesort operates by advancing through the sequence S, swapping adjacent values S[i] and S[i +1] if S[i] < S[i + 1]. In the worst case, 16 passes through a sequence of size 16 needs to be done.

The correct output for 16 iterations is:

```
61
87
154
170
275
426
503
509
512
612
653
677
703
765
897
908
```

Instead, the code is producing the following output:

```
503
87
503
61
503
170
503
275
503
426
154
503
503
503
503
503
```

```
void->void pipeline BubbleSort {
    int SIZE = 16;
    int[SIZE] data;

    data[0] = 503; data[1] = 087;
    data[2] = 512; data[3] = 061;
    data[4] = 908; data[5] = 170;
    data[6] = 897; data[7] = 275;
    data[8] = 653; data[9] = 426;
    data[10] = 154; data[11] = 509;
    data[12] = 612; data[13] = 677;
    data[14] = 765; data[15] = 703;

    add IntSource(SIZE, data);
    add BubbleSortPipe(SIZE);
    add IntPrinter();
}

// An input of SIZE random numbers to
// sort.
void->int filter IntSource(int SIZE,
    int[SIZE] data) {
    int index = 0;
    init {
    }
    work push 1 {
        push(data[index++]);
        if (index == SIZE) {
        index = 0;
        }
    }
}

int->int pipeline BubbleSortPipe(int
    SIZE) {
    for(int i = 0; i < SIZE; i++) {
        add Bubble(SIZE);
    }
}
```

```
int->int filter Bubble(int SIZE) {
    int max;
    int index;
    init {
        index = 0;
    }
    work push 1 pop 1 peek 2 {
        if (index == 0) {
            max = peek(0);
        }

        if (index == SIZE - 1) {
            push(max);
        } else {
            int current = peek(0);
            if (max > current) {
                push(current);
            } else {
                push(max);
                max = current;
            }
        }
        index = (index+1) % SIZE;
        pop();
    }
}

int->void filter IntPrinter() {
    init {
    }
    work pop 1 {
        print(pop());
    }
}
```

A.4.8 Application 8: BitReverse

The purpose of the code on the next page is to sort a sequence of 16 consecutive numbers in bit-reversed order. A sequence of numbers is in bit-reversed order when reversing the bits of each number results in a sequence of consecutive numbers. For example, the sequence

```
000,  100,  010,  110,  001,
101,  011,  111
```

is in bit-reversed order because reversing the bits of each number results in a sequence of consecutive numbers:

```
000,  001,  010,  011,  100,
101,  110,  111
```

Notice in the code on the next page that the numbers are represented as integers and the output is in decimal format (not binary).

The correct output for 1 iteration is:

```
0
4
2
6
1
5
3
7
```

Instead, the code is producing the following output:

```
0
4
0
4
2
6
2
6
1
5
1
5
3
7
3
7
```

```
void->void pipeline BitReverse {
    int N = 8;
    add IntSource(N);
    add Reorder(N);
    add IntPrinter();
}

int->int splitjoin SplitjoinReorder(int
    N) {
    split roundrobin;
    add Reorder(N/2);
    add Reorder(N/2);
    join roundrobin(N);
}

int->int pipeline Reorder(int N) {
    if (N > 2) {
        add SplitjoinReorder(N);
    } else {
        add Pass(N);
    }
}

int->int filter Pass(int N) {
    init {
    }
    work push N pop N {
        for (int i = 0; i < N; i++) {
            push(pop());
        }
    }
}

void->int filter IntSource(int N) {
    int x;
    init {
        x = 0;
    }
    work push 1 {
        push(x++);
        if (x == N) {
            x = 0;
        }
    }
}

int->void filter IntPrinter {
    init {
    }
    work pop 1 {
        print(pop());
    }
}
```

A.4.9 Application 9: Overflow

The code for this application (which can be found in your workspace) is a nonsensical program that is generating a NaN in the first iteration:

```
128.54594
126.08129
118.08861
117.43733
116.096664
115.08876
114.013054
112.93948
111.85341
110.75725
110.742065
110.72697
110.69963
110.68819
110.6757
110.66378
110.65177
110.639786
110.62779
110.6158
NaN
NaN
NaN
...
```

NaN stands for "not a number" and can be obtained, for example, by dividing 0/0. Instead of debugging the entire application, your task is to comment out exactly one line that gives the first NaN, so that the program will not output a NaN for at least the first 1000 iterations.

A.5 Post-Study Questionnaire

Please complete the below questionnaire.

- Check or uncheck a box ☐ by clicking on it.
- Select from a drop-down box 1 (easy) by clicking on it and selecting from the drop-down list box that appears. The default selection is "1 (easy)."
- Type into a text field by clicking on the field and typing. The text field expands as you type. You can type multiple lines of text by pressing enter.

When you are finished, please save the file as

C:\Program Files\eclipse\workspace\Post-Study Questionnaire.**doc**

by going to the **File** menu and clicking **Save As...** (notice that the file currently ends in **dot**).

1. For each application, please rate the difficulty of debugging the application and give a general description of how you went about debugging each application.

Application	Difficulty (1 is easy, 10 is hard)	Description
1	1 (easy)	
2	1 (easy)	
3	1 (easy)	
4	1 (easy)	
5	1 (easy)	
6	1 (easy)	
7	1 (easy)	
8	1 (easy)	
9	1 (easy)	

2a. For the applications in which you used the SDT, did you find any bugs in the SDT itself?

☐ Yes ☐ No

2b. If yes, please list them below.

3. For the applications in which you used the SDT, please list any problems you had with using the SDT.

4a. Did you find the SDT easy or hard to learn? 1 (easy)

4b. Why did you find the SDT easy or hard to learn?

5a. Did you find the SDT slow or fast to use after you had learned it? 1 (slow)

5b. Why did you find the SDT slow or fast to use after you had learned it?

6a. Was it easy or hard to remember what you had learned? 1 (easy)

6b. Why was it easy or hard to remember what you had learned?

7a. Would you use this SDT for debugging StreamIt applications? ☐ Yes ☐ No

7b. Why would you or would you not use this SDT for debugging StreamIt applications?

8a. Did you find the SDT easy or hard to use? 1 (easy)

8b. Please list any functionality of the SDT that made debugging easy or hard.

9. Was there any functionality you wish the SDT had?

10. Please write any comments you have related to the SDT.

11. Please suggest any improvements you have to the SDT.

Bibliography

[1] Simulink 5.1.1. http://www.mathworks.com/products/simulink.

[2] Matlab 6.5.1. http://www.mathworks.com/products/matlab.

[3] H. Agrawal, R.A. DeMillo, and E.H. Spafford. Efficient Debugging with Slicing and Backtracking. *Software Practice and Experience*, pages 589–616, June 1993.

[4] T. Akgul, P. Kuacharoen, V.J. Mooney, and V.K. Madisetti. A Debugger RTOS for Embedded Systems. Technical report, Georgia Institute of Technology, School of Electrical and Computer Engineering, Atlanta, Georgia.

[5] Advanced Computer Organization: Stream Processor Architecture. http://cva.stanford.edu/ee482s.

[6] BoxView. http://www.domaintec.com/BoxView.html.

[7] RDS Software Development Kit (SDK) Product Brief. http://www.cradle.com/products/tools/development_tools-software.shtm.

[8] C. Caerts, R. Lauwereins, and J.A. Peperstraete. PDG: A Process-Level Debugger for Concurrent Programs in the GRAPE Parallel Programming Environment. Technical report, Katholieke Universiteit Leuven, E.S.A.T. Laboratory, Heverlee, Belgium.

[9] Debug-56K. http://www.domaintec.com/Debug56K.html.

[10] Analog Devices, Inc. http://www.analog.com.

[11] Eclipse 2.1 Documentation. http://help.eclipse.org/help21/index.jsp.

[12] The MULTI® Integrated Development Environment. http://www.ghs.com/products/MULTI_IDE.html.

[13] Controlled Experiments. http://graphics.lcs.mit.edu/classes/6.893/F03/lectures/L13.pdf.

[14] B. Hailpern and P. Santhanam. Software Debugging, Testing, and Verification. *IBM Systems Journal*, 41(1):5–8, November 2002.

[15] Advanced Topics in Computer Architecture: Chip Multiprocessors & Polymorphic Processors. http://www.stanford.edu/class/ee392c.

[16] Object Technology International, Inc. Eclipse Platform Technical Overview. Technical report, IBM Corporation, February 2003.

[17] Research Issues. http://pages.cpsc.ucalgary.ca/saul/681/1997/jas/issues.html.

[18] G. Jongren. How to Learn DSP Programming. March 2004.

[19] Momentics®. http://www.qnx.com/products/development/momentics_glance.html.

[20] Screenshot of the MULTI® Integrated Development Environment. http://www.ghs.com/products/images/MULTI_browser_big.gif.

[21] CS 252 Spring 2002 Project on Streams. http://www.cs.berkeley.edu/nmani/courses/cs252/proposal.html.

[22] Eclipse 2.1.1 Download Page. http://download2.eclipse.org/downloads/drops/R-2.1.1-200306271545/index.php.

[23] Eclipse Home Page. http://www.eclipse.org.

[24] Eclipse Platform Subproject Page. http://www.eclipse.org/platform/index.html.

[25] StreamIt Development Tool Page. http://catfish.csail.mit.edu/streamit/html/eclipse-plugin.html.

[26] StreamIt Documentation Page. http://catfish.csail.mit.edu/streamit/html/documentation.html.

[27] StreamIt Download Page. http://catfish.csail.mit.edu/streamit/html/download.html.

[28] StreamIt Home Page. http://catfish.csail.mit.edu/streamit.

[29] StreamIt Research Page. http://catfish.csail.mit.edu/streamit/html/research.html.

[30] R.J. Ridder. Programming Digital Signal Processors with High-Level Languages. *DSP Engineering*, 2000.

[31] S. Saba. Embedded Digital Technology Defines the Next Generation of Medical Electronics. In *Medical Electronics Manufacturing*, 1999.

[32] Embedded Workbench™ Screenshot. http://www.iar.com/FilesPublic/EW/000319/gen3ide.gif.

[33] Momentics® Screenshot. http://www.qnx.com/popups/imageview.html?group=screenshot&key=src_debugger.

[34] R. Stephens. A Survey of Stream Processing. *Acta Informatica*, 34(7), 1997.

[35] IAR Systems. http://www.iar.com.

[36] QNX Software Systems. http://www.qnx.com.

[37] Cradle Technologies, Inc. RDS Software Development Kit (SDK) Product Brief. Technical report.

[38] Cradle Technologies, Inc. http://www.cradle.com.

[39] Domain Technologies, Inc. http://www.domaintec.com.

[40] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A Language for Streaming Applications. In *Proceedings of the International Conference on Compiler Construction (CC)*, 2002.

[41] Java Development Tooling. `http://www.eclipse.org/jdt/index.html`.

[42] VisualDSP++. `http://www.analog.com/Analog_Root/static/technology/dsp/beginnersGuide/quickguide2.html`.

[43] E.L. Waingold. SIFt: A Compiler for Streaming Applications. Master's thesis, Massachusetts Institute of Technology, June 2000.

[44] Embedded Workbench^{TM}. `http://www.iar.com/Products/EW`.