A GRAPHICAL USER INTERFACE FOR COMPOSING PARALLEL COMPUTATION IN THE STAPL SKELETON LIBRARY

An Undergraduate Research Scholars Thesis

by

ABBY MICHELLE MALKEY

Submitted to the Undergraduate Research Scholars program
Texas A&M University
in partial fulfillment of the requirements for the designation as an

UNDERGRADUATE RESEARCH SCHOLAR

Approved by
Research Advisor: Dr. Nancy M. Amato

May 2016

Major: University Studies, College of Architecture

TABLE OF CONTENTS

ABSTRACT

A Graphical User Interface for Composing Parallel Computation in the STAPL Skeleton Library

Abby Michelle Malkey
College of Architecture
Texas A&M University

Research Advisor: Dr. Nancy M. Amato
Department of Computer Science and Engineering

Parallel programming is a quickly growing field in computer science. It involves splitting the computation among multiple processors to decrease the run time of programs. The computations assigned to a processor can depend on the results of another computation. This dependence introduces a partial ordering between tasks that requires coordination of the execution of the tasks assigned to each processor. OpenMP and MPI are current heavily utilized approaches and require the use of low level primitives to express very simple scientific applications. Newer environments, such as STAPL [15], Charm++ [2], and Chapel [1], among others, raise the level of abstraction, but the challenge of specifying the flow of data between computations remains. However, graphical user interfaces (GUIs) can simplify this task. The purpose of this project is to create a GUI that allows a user to specify a parallel application written in STAPL by composing high-level components and by defining the flow of data between them. The idea is that the user creates the layout of the code using shapes and lines, which produce the composition on an underlying layer, eliminating the need to write complex composition specifications directly in the code.

DEDICATION

I dedicate this research to my parents, for putting up with me and supporting the many endeavors upon which I choose to embark.

ACKNOWLEDGMENTS

Many people have enabled the completion of this thesis. Thanks to Nancy Amato and Timmie Smith for guidance and support throughout the duration of this project. I would especially like to thank Mani Zandifar for all of his help and patience that enabled me to complete this work.

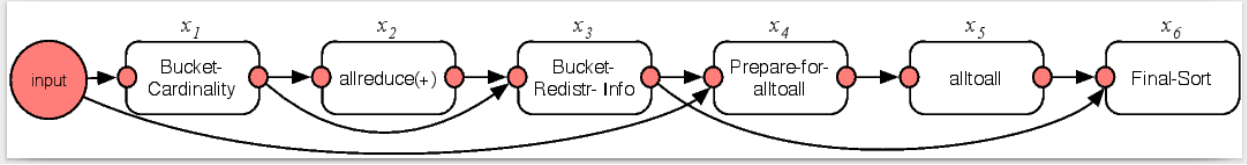
CHAPTER I

INTRODUCTION

Parallel programming is a complicated process. When computations are split among processors, dependencies can arise are difficult to represent in code. Some current approaches are very low-level, but newer methods are more abstract and easier to use. However, they still lack a simple way to specify the flow of data between computations. This can be simplified by creating a graphical interface to abstract some of the more tedious and low level features of creating skeletons by eliminating the need to write complex code specifications. An example is the NAS IS composition, shown below. In this example, *N* keys are sorted, and then uniformly distributed in memory and generated using a predefined sequential key generator. First, the range of possible input values are put into buckets, then each partition counts the number of values in each bucket. The total number of elements in each bucket is then computed and made globally known to all partitions. A partitioning of buckets is devised with the knowledge of the key distribution. The keys are prepared for a global exchange, then redistributed to the partitions. Lastly, each partition sorts the keys it received from any other partition [9]. As can be seen in the second diagram below, there are overlapping dependencies between different elements of the program.

```
skeletons::compose<skeletons::tags::inline_flow>(
  (x0= map(bucket_cardinality(traits.num_buckets(), shift)) )[keys_in],
   x1 = allreduce<tags::right_aligned>(vec_plus_t()), //use default, x1<-x0
   x2 = map(bucket_and_element_offset(keys_array_size, n_locs)), //x2 <- x1
  (x3= zip<3>(prepare_for_alltoall<alltoall_val_t>(traits.num_buckets(),
                                                   shift)) )[x2,x1,keys_in],
   x4 = alltoall<alltoall_val_t, tags::ALLTOALL_TAG>(), //v4 <- v3
  (x5 = zip<3>(final_sort(
               loc_id, traits.num_buckets(),
               traits.max_key() / traits.num_buckets())))[x4, x2, ranks_in]
);
```

A code snippet from the is_skeletons file.



A diagram of the NAS benchmark.

CHAPTER II

METHODOLOGY

Editor Evaluation

Different existing editors were sampled for use in this project. Eclipse GEF has a logicEditor, but the interface was far too complicated to manipulate to have the desired results. NoFloJS not only didn't contain all of the functionality needed, but the process to download, install, and configure it was far too involved and lengthly to be user-friendly. The current environment that will be used to complete the project is an EMF model through Eclipse. It takes a model file and generates controller classes that are used to create views, from which skeletons are composed.

Skeleton composition application model

The application is specified with an Ecore model. The model describes the classes that implement the behaviors of the GUI. These features include creating shapes for each skeleton and drawing lines to express the relationship between them. The genmodel file uses the information in the Ecore file to generate the appropriate Java class files that implement the behavior of the components. The editor itself depends on Java and Eclipse plug-ins and runs in a separate Eclipse window as its own application. Editor classes are written and added as extensions to the main plug-in. The views and skeletons that represent the data and computation of a parallel program are basic 2D shapes drawn in the editor when the program is running. Classes are given default values, if applicable, when defined and object types are given ranges or restraint for data, if necessary. There is also a root edit part that creates a canvas upon which these views are drawn. The model that creates the edit parts is loaded in the editor and displayed. The views and skeletons (shapes) can be created from the palette in the editor and given a user-defined location and name.

Skeleton composition application

The program generated from the model runs as a standalone Java application in a new Eclipse window. From this window, there is a blank canvas window in which the Composite diagrams

are created. To the right of this window, there is a Palette menu from which the user can select Skeletons or Views to create and the different types of links to connect them (Input, Output, Flow). These links are distinguished by the design on their endpoints. Above the Palette, there are undo, redo, and delete buttons that are applicable to actions performed in the main window. Some actions, like delete, must be performed on selected objects. Objects may be selected by clicking on them in the main window or by clicking and dragging to create a box that selects multiple objects at a time. Each object has points around its perimeter that can be used to grow or shrink the object by clicking and dragging. By clicking and dragging in the center of the object, the entire object itself can be moved anywhere within the main window. For objects that are connected to other objects by links, the links will grow and shrink with respect to the changing location of the object. Each object has a name beneath it, which can be changed by clicking on the existing name to open a text box. Upon closing the window, a dialog box pops up with the option to save existing changes.

Skeleton files

When figures are drawn in the editor, an XML file describing their layout and composition is produced. This can be translated to JSON using existing methods and then to STAPL code for use in STAPL applications.

CHAPTER III

IMPLEMENTATION

Some of the key classes used in this implementation are discussed below. These classes implement the core capabilities of the application.

CompositeGraphicalEditor

This class is an extension of the Java class GraphicalEditorWithFlyoutPalette. It provides functions to construct the editor, initialize its contents, prepare it to hold the Edit Parts (the figures drawn in the editor), and create the palette from which the components are selected. It also has functions to save the status of the program and outputs an XML file representing the layout.

CompositeEditPartFactory

The CompositeEditPartFactory class determines what type of Edit Part to create in the editor based on which type of Part is selected in the palette and inserts it in the editor frame.

SkeletonFigure

This class creates a Skeleton, which consists of a Rectangle shape and a rectangular name label and binds it to the editor with a Connection Anchor. The orginal Rectangle has a default size, but can be resized later by the user. The Skeleton is created wherever the mouse is clicked in the editor window.

ViewFigure

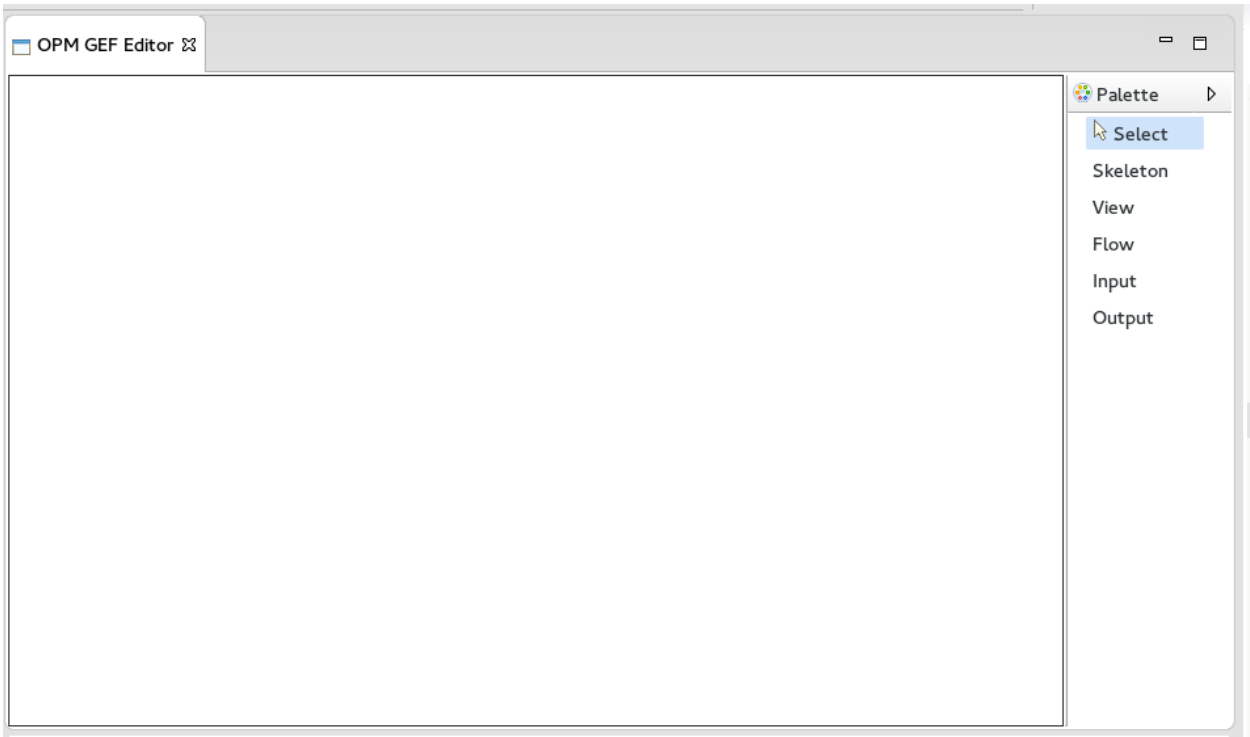
ViewFigure behaves like the SkeletonFigure class, except it handles Views, which represent data in the parallel application. It creates a standard View in the appropriate location and binds it to the editor.

FlowEditPart

This class is a specified implementation of the AbstractConnectionEditPart Java class. It creates a Flow Adapter, and gives it the endpoints as indicated by the user connecting two objects. Should the objects with Flow connections be moved, the Flow Part is able to adjust accordingly by extending, contracting, and bending, and updating to reflect these changes visually.

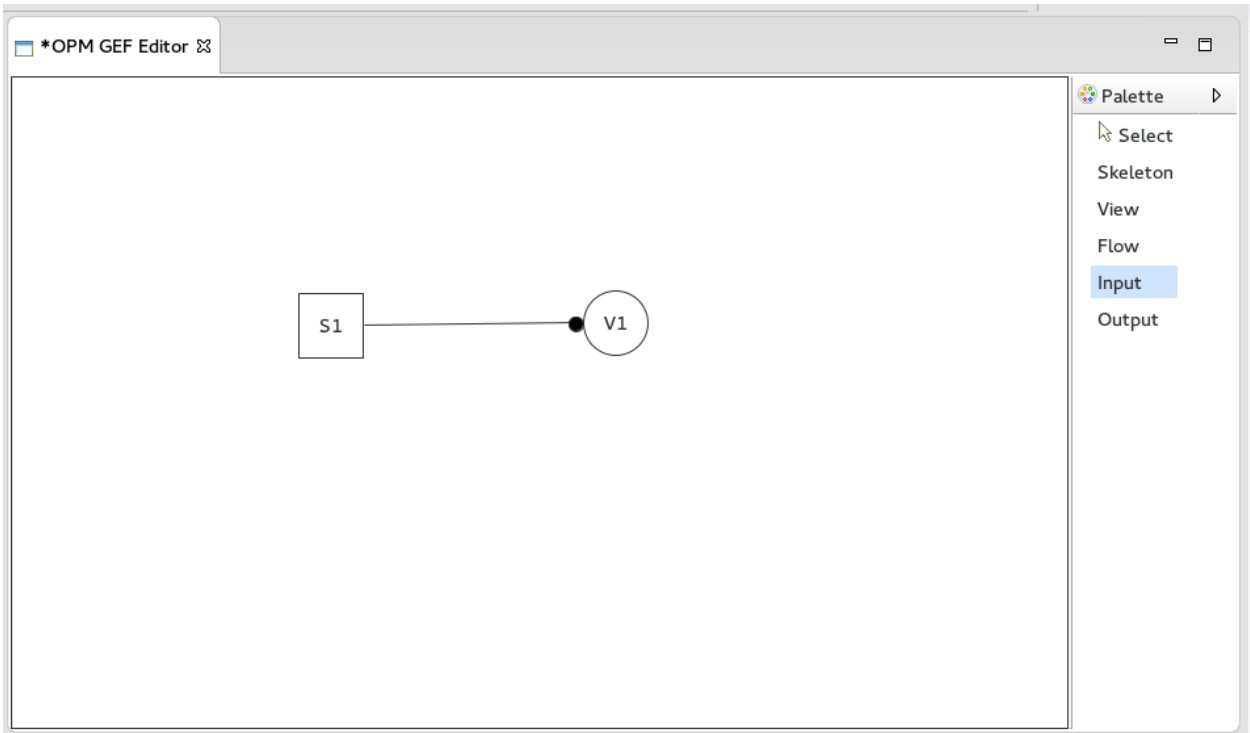
CHAPTER IV

INTERFACE DEMONSTRATION

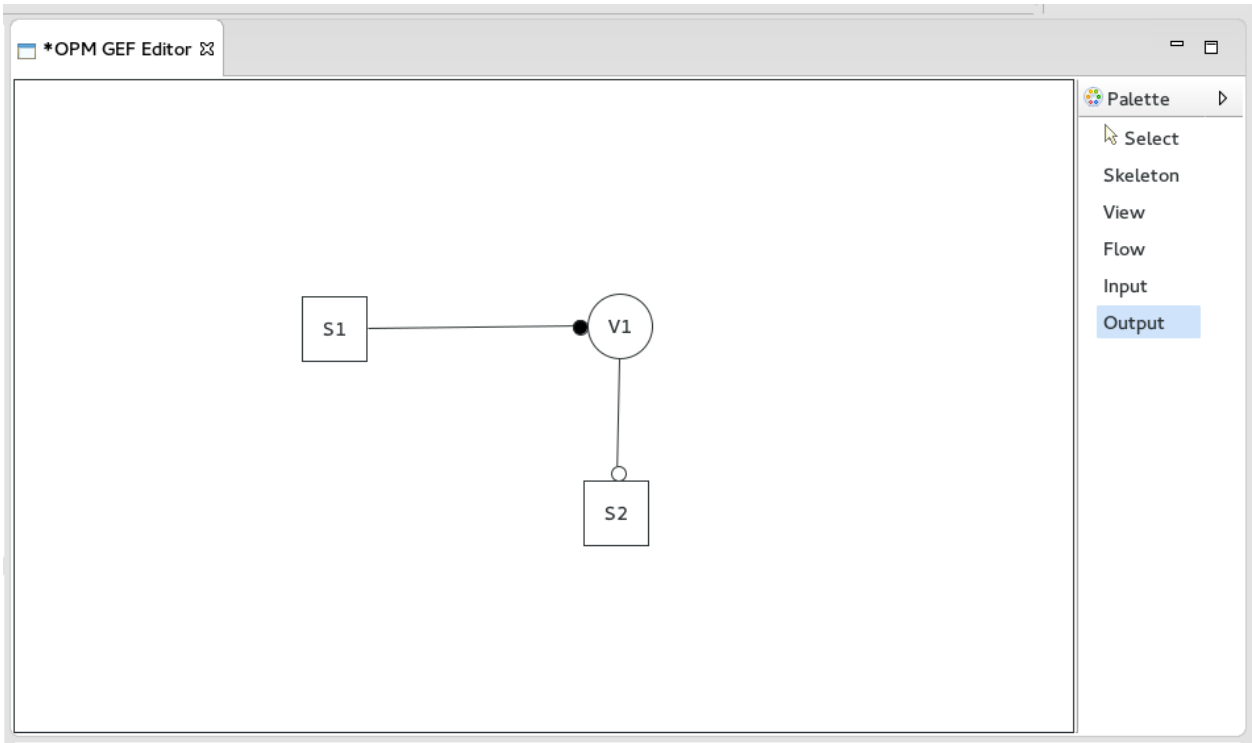


Running the source code opens a new Eclipse application. Opening a new project and Diagram results in the default editor window.

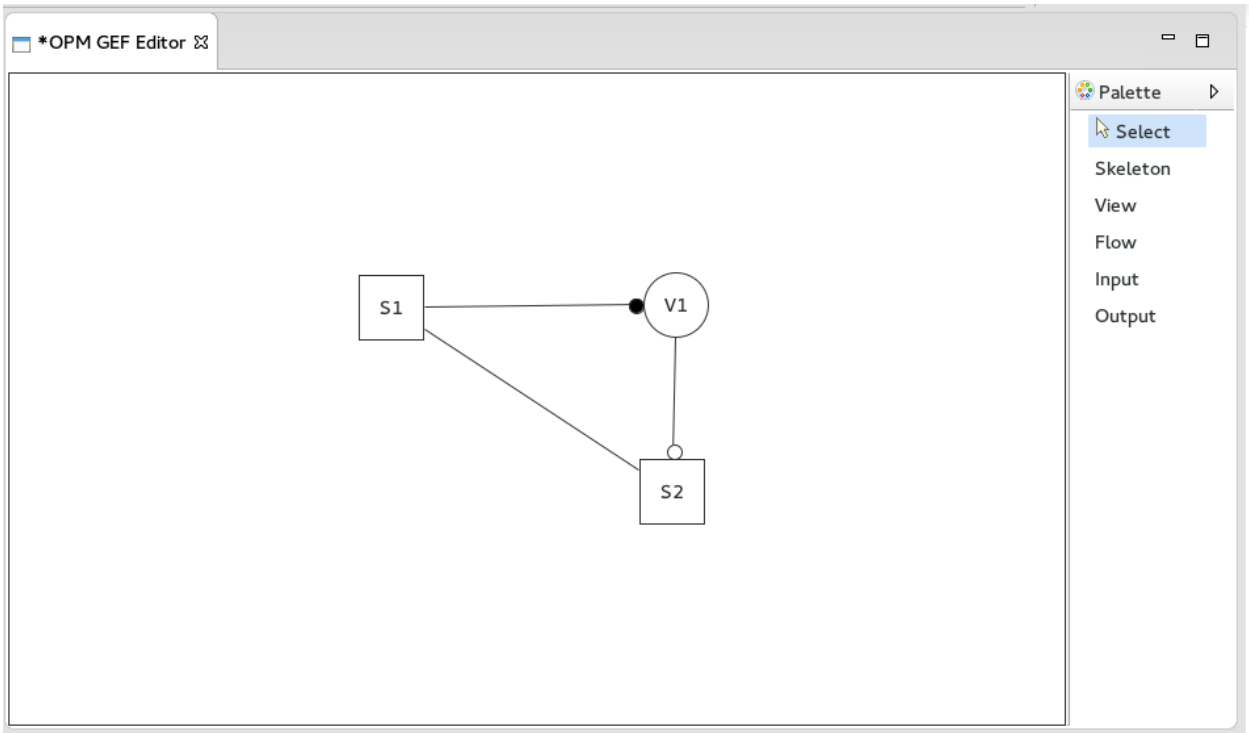
A Skeleton (S1) and a View (V1) are created from the palette to the right and placed upon clicking anywhere in the editor.



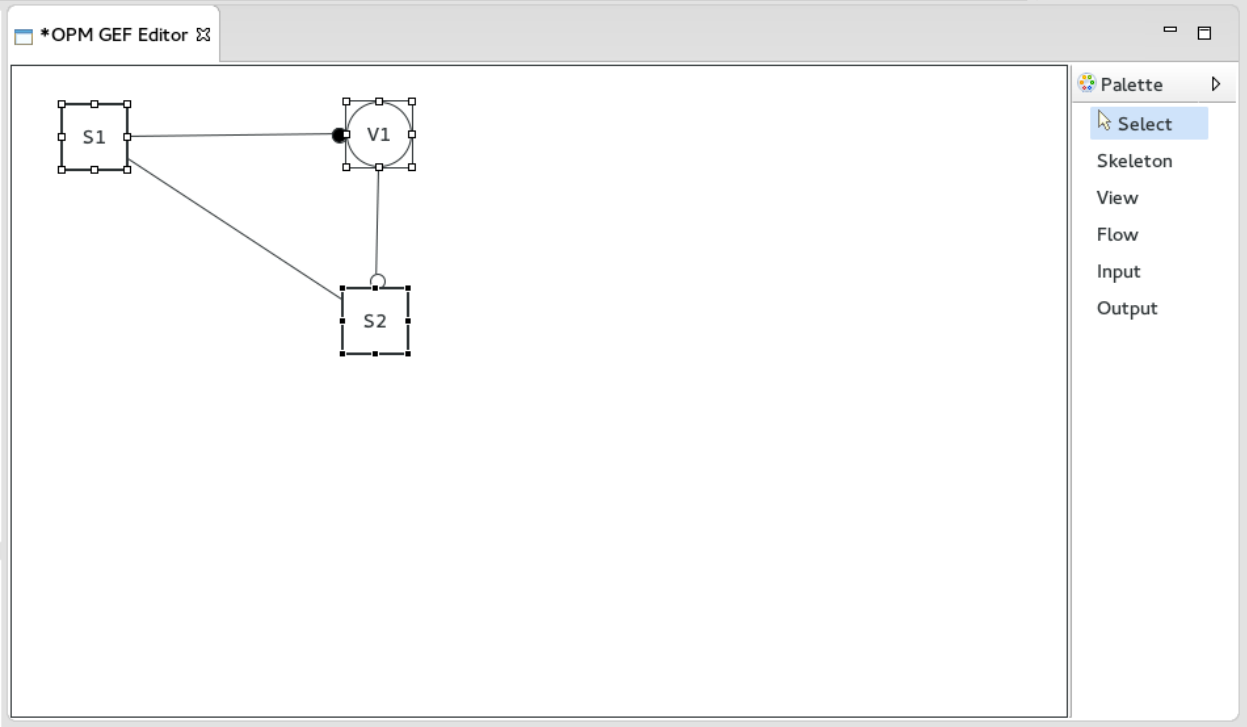
An Input link is placed between S1 and V1.



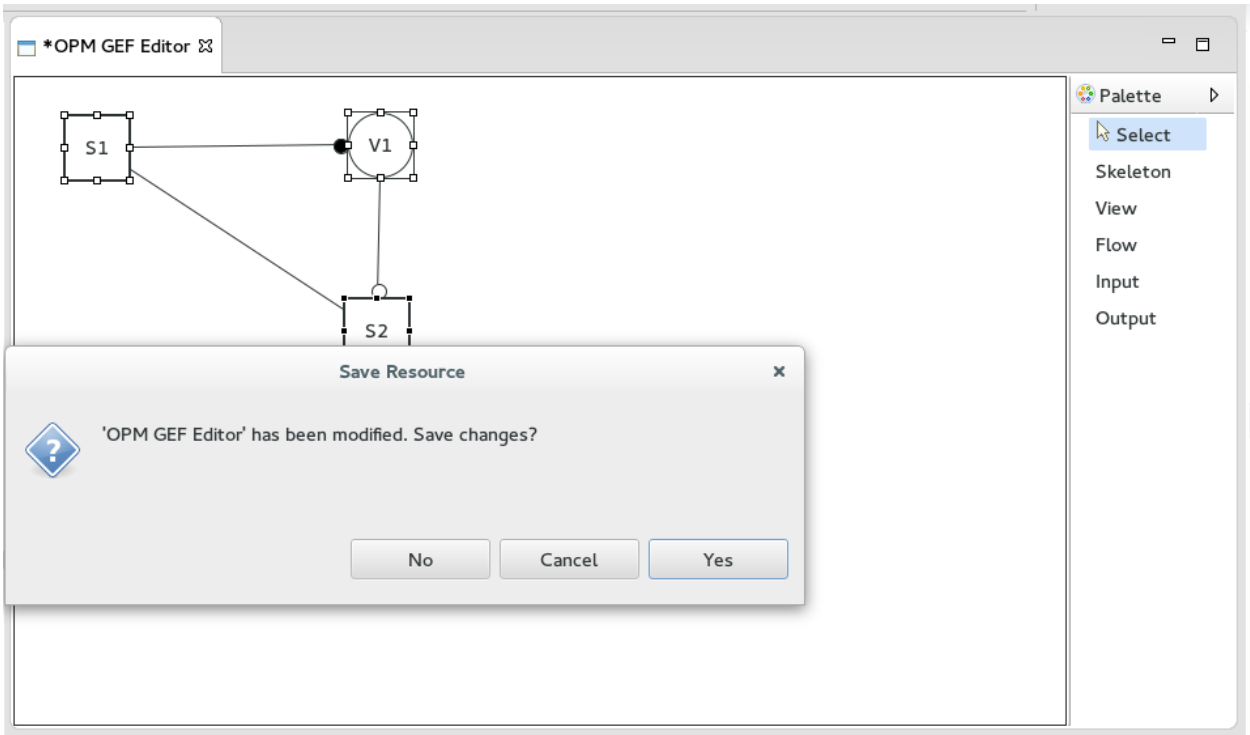
A second Skeleton (S2) is created and placed in the editor, and an Output link is placed between V1 and S2.



A multi-directional Flow is placed between S2 and S1.



The entire composition has been selected and moved to the upper left corner of the editor.



Upon attempting to close the Eclipse window, the option to save the status of the editor appears.

CHAPTER V

FUTURE WORK

The next step of this project is to further develop the editor to allow the creation of a complete representation of the NAS benchmark which will produce XML code. Afterwards, the XML code needs to be converted to JSON for further integration into existing STAPL code. Many methods are currently available to do this, and tools also exist to generate C++ code from a JSON specification.

REFERENCES

[1] Bradford L. Chamberlain, David Callahan, Hans P. Zima. *International Journal of High Performance Computing Applications*, August 2007, 21(3): 291-312.

[2] L. Kale, A. Arya, N. Jain, A. Langer, J. Lifflander, H. Menon, X Ni, Y. Sun, E. Totoni, R. Venkataraman, L. Wesolowski. *Charm++: Migratable Objects + Active Messages + Adaptive Runtime = Productivity + Performance.* Parallel Programming Laboratory, Department of Computer Science, University of Illinois at Urbana-Champaign.

[3] P. Newton, J. Browne. *The CODE 2.0 Graphical Parallel Programming Language.* Department of Computer Science, University of Texas at Austin, pp. 167-177.

[4] P. E. Haeberli. *ConMan: a visual programming language for interactive graphics.* In: Proceedings ACM SZGGRAPH 88. (1988) Printed as ACM Computer Graphics 22 August, 103-111.

[5] F. Ludolph, Y. Y. Chow, D. Ingalls, S. Wallace, K. Doyle. *The Fabrik programming environment.* IEEE Workshop on Visual Languages (1989). Pittsburgh, Pennsylvania, October 10-12, pp. 222-230.

[6] D. Hils. *Visual Languages and Computing Survey: Data Flow Visual Programming Languages.* Journal of Visual Languages and Computing (1992) 3, 69-101.

[7] W. Johnston, J. Hanna, R. Millar. ACM Computing Surveys, Vol. 36, No. 1, March 2004, pp. 1-34.

[8] *LabVIEW: a demonstration.* National Instruments Corporation (1987) National Instruments Corp., 12109 Technology Blvd., Austin, Texas 78727-6204.

[9] M. Zandifar, N. Thomas, N. M. Amato, L. Rauchwerger. *The STAPL Skeleton Framework.* Parasol Lab, Dept. of Computer Science, Texas A&M University. 2014.

[10] M. Zandifar. *Abstract Reduction Operation Models in the LIME Programming Model.* Parallel and Distributed Systems Group, Department of Software Technology, Delft University of Technology, Delft, The Netherlands. June 2009.

[11] S. Matwin, T. Pietrzykowski. *PROGRAPH: A preliminary report.* (1985) Computer Languages 10,91-126.

[12] M. Zandifar, N. Thomas, N. M. Amato, L. Rauchwerger. *The Stapl Skeleton Framework.* Languages and Compilers for Parallel Computing - 27th International Workshop, LCPC 2014, Hillsboro, OR, USA, September 15-17, 2014, Revised Selected Papers pp. 176-190. 2014.

[13] M. Zandifar, M. A. Jabbar, A. Majidi, D. Keyes, N. M. Amato, L. Rauchwerger. *Composing Algorithmic Skeletons to Express High-Performance Scientific Applications.* Proceedings of the 29th ACM on International Conference on Supercomputing, ICS'15, Newport Beach/Irvine, CA, USA, June 08 - 11, 2015. pp. 415-424. 2015.

[14] D. Skillicorn, D. Talia. *Models and Languages for Parallel Computation.* ACM Computing Surveys, Vol. 30, No. 2, June 1998.

[15] A. Buss, Harshvardhan, I. Papadopoulos, O. Pearce, T. Smith, G. Tanase, N. Thomas, X. Xu, M. Bianco, N. M. Amato, and L. Rauchwerger. *Stapl: Standard Template Adaptive Parallel Library.* In Proceedings of the 3rd Annual Haifa Experimental Systems Conference, SYSTOR 10, pages 14:114:10, New York, NY, 2010. ACM.

[16] M. Toyoda, B. Shizuki, S. Takahashi, S. Matsuoka, E. Shibayama. *Supporting Design Patterns in a Visual Parallel Data-flow Programming Environment.* Department of Mathematical and Computing Sciences, Tokyo Insitute of Technology, Tokyo, Japan. IEEE 1997.

[17] Vainolo. *Creating a GEF editor.* http://www.vainolo.com/2011/06/12/creating-a-gef-editor-part-1-defining-the-model-2/