ABSTRACT

Lavana, Hemang. A Universally Configurable Architecture for Taskflow-Oriented Design of a Distributed Collaborative Computing Environment
(Under the direction of Dr. Franc Brglez.)

The Internet continues to change the models of how we organize shopping, business transactions, learning, entertainment, domain-specific projects and work in general. A web-browser is a relatively simple entry-point to access tools and services that support a number of such activities. Expectations are rising to do more. Shoppers can already customize and aggregate the on-line bank and investment accounts. Industries such as chip design and biotechnology contract with compute farms for customized and scalable high performance solutions. We argue that the potential of the Internet can be explored further once the user is provided with significant choices to configure the computing environment on her own. Such an environment provides a number of choices not only for an individual user to engage in a project with distributed computing resources, but also for a distributed team to engage in collaborative projects.

The major goal of this thesis is to create a powerful distributed computing environment and render it collaborative. A distributed computing environment supports four structures that create a program as a hierarchy of tasks: (1) task sequencing, (2) decisions, allowing data to control the task sequence, (3) iterations, repeating the same task sequence multiple times, (4) encapsulation, replacing a group of tasks with a name that denotes the group. In addition, the environment is rendered asynchronously collaborative by (1) ownership and locking of data, (2) ownership and authorized execution of tasks, and synchronously collaborative by (1) negotiated sharing of task sequencing, (2) coordinated execution of the entire sequence of tasks.

The distributed computing environment is implemented by way of a user configurable universal client (OmniFlow). The key to user configurability of the client is the simplicity of the underlying taskflow model that relies on user-defined compositions of two component types only: a blackbox component and a whitebox component, each
encapsulated with an autonomous FSMD, (finite-state-machine with a datapath), with the blackbox/whitebox component representing an extension of the datapath itself. A task instance layer of such a component is formed by always accessing the encapsulated layer through a ControlJoin and ControlFork primitive. The interconnection of such task instances forms a taskflow or a whitebox component. A taskflow is an intersection of a task graph and data graph where the task graph is a directed acyclic graph, with two distinguished vertices, a source and a sink, where the source represents the first task (BeginFork), and the sink represents the last task (EndJoin). The task graph is thus a polar DAG of interacting asynchronous FSMDs with built-in synchronizing primitives that can be executed concurrently and readily synchronized, just as would any DAG of well-designed and well-matched hardware components. Encapsulated tasks and task instances are specified using XML schema and a collaborative distributed task mark-up language (cdtML).

We propose a new approach, based on a hybrid architecture, to render any single-user Tcl application collaborative. This approach consists of a centralized synchronous group server (SGS) and a distributed synchronous group client (SGC), both are written in Tcl, and are universal in that any Tcl application interfaced to SGC will be rendered collaborative. Specifically, we rely on SGC to read static configuration files, easily created and stored, which define the collaborative behavior that the Tcl application is to assume. In addition, SGC provides an Inter-client synchronization table that allows the user to dynamically change the collaborative behavior of the application during runtime. This is important when addressing different user preferences and the dynamically changing needs of collaborative sessions.

Several experimental projects, devised with OmniFlows, demonstrate the utility and versatility of the OmniFlow environment by encapsulating a wide variety of distributed components, including commercial and university-based tools. In addition, experimental evaluations of the prototype implementation, consisting of up to 9150 tasks and a longest path of 1600 tasks, not only reveals the scalability of the environment, but also an exceptionally good asymptotic performance of the task scheduler, thereby demonstrating the overall effectiveness of the proposed architecture.
A Universally Configurable Architecture
for Taskflow-Oriented Design of a
Distributed Collaborative Computing Environment

by

Hemang Lavana

A thesis submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Department of Electrical and Computer Engineering

Raleigh, N.C.

December, 2000

Approved By:

__________________________  __________________________
Dr. Mladen Vouk (Chair)     Dr. Franc BrGLEZ (Co-Chair)

__________________________  __________________________
Dr. Munindar Singh          Dr. Matthias Stallmann
Biography

Hemang Lavana received his Bachelor’s (1992) from SGGS College of Engg. and Tech., Maharashtra, India and his Master’s (1994) from Indian Institute of Technology, Bombay, India. He worked on design and development of FPGA chips for one year in JISL, Bombay. He joined the PhD program in the Department of Electrical and Computer Engineering, North Carolina State University, Raleigh in Fall 1995. Since summer 1999, he is also working on design and development of a modular, scalable, distribution mechanism for ATS products with Cisco Systems.

His research interests include Web-based VLSI design and collaborative, distributed programming. He has published about 9 papers in various conference publications.
Acknowledgments

I express my deepest gratitude to Dr. Franc Brglez, who serves as the Director of CBL (Collaborative Benchmarking Laboratory). Dr. Brglez provided invaluable guidance and supervision throughout the research of this thesis and contributed several hours of research discussions. His extensive editing sessions have greatly improved the presentation of this thesis.

I thank Dr. Mladen Vouk, Dr. Munindar Singh, Dr. Matthias Stallmann and Dr. Felix Wu for serving on my thesis committee. Dr. Vouk, Dr. Singh and Dr. Wu’s comments and feedback at various stages of this research have been very valuable. Dr. Stallmann, who became part of the advisory committee only in its final stage, has also provided useful comments throughout this research.

I also thank Dr. Robert Reese, Mississippi State University, for providing the access to JavaCADD tools and Dr. Gershon Kedem, Duke University, for providing Dr. Brglez the access to a remote testbed server.

My colleagues at the Collaborative Benchmarking Laboratory, Deabrata Ghosh, Amit Khetawat, Nevin Kapur, Adel ElMessiry, Justin Harlow, Andrej Zemva, Roman Kuznar, Kris Kozminsli and Bruce Deuwer, and also my friends, Yasho Potlapalli, Santosh Kolenchery, Tarun Jain, Sirkanth Mahalingm and Sanjukta Ghosh have my sincerest appreciation for their support throughout my stay. I also thank for all the help and encouragement I received from Lucy Ringland, research technician at CBL, as well as the faculty, students, and staff at the ECE and CS department.

I gratefully acknowledge DARPA, SEMATECH, and Semiconductor Research Corporation for supporting me throughout my course work and research.

I also acknowledge Jay Yang, manager of ATS group with Cisco Systems, for supporting me and accommodating a flexible work schedule to complete my thesis.

Finally, I would sincerely like to thank Dhiraj Thakkar and Neelima Lavana, who have constantly encouraged me to pursue for higher education throughout my life and without whom I would not have been able to study this far. I would also like to thank my parents and my family members for their love, encouragement and patience throughout my studies.
## Contents

<table>
<thead>
<tr>
<th>List of Figures</th>
<th>vii</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>1</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>2</td>
</tr>
<tr>
<td>1.2 Motivation</td>
<td>7</td>
</tr>
<tr>
<td>1.3 Thesis Goals</td>
<td>11</td>
</tr>
<tr>
<td>1.4 Thesis Outline</td>
<td>16</td>
</tr>
<tr>
<td>2 Related Work</td>
<td>19</td>
</tr>
<tr>
<td>2.1 Workflow Technologies</td>
<td>19</td>
</tr>
<tr>
<td>2.1.1 WorldFlow - Virtual Warehouses using Workflows and the Web</td>
<td>20</td>
</tr>
<tr>
<td>2.1.2 PUNCH - Purdue University Network Computing Hubs</td>
<td>21</td>
</tr>
<tr>
<td>2.1.3 WELD - An Environment for Web-based Electronic Design</td>
<td>22</td>
</tr>
<tr>
<td>2.1.4 WorkXpert - A Process Management Tool</td>
<td>24</td>
</tr>
<tr>
<td>2.1.5 JavaCADD - Java-based Distributed ECAD Services</td>
<td>25</td>
</tr>
<tr>
<td>2.1.6 FlowMake - A Workflow Modeling and Verification Tool</td>
<td>26</td>
</tr>
<tr>
<td>2.2 Computer Supported Cooperative Work</td>
<td>27</td>
</tr>
<tr>
<td>2.2.1 VNC - A Virtual Network Computing Environment</td>
<td>28</td>
</tr>
<tr>
<td>2.2.2 GroupKit - A Groupware Applications Development Toolkit</td>
<td>29</td>
</tr>
<tr>
<td>2.2.3 CBE - A Collaboratory Builder’s Environment</td>
<td>30</td>
</tr>
<tr>
<td>2.2.4 TANGO - A Collaborative Environment for the World-Wide Web</td>
<td>31</td>
</tr>
<tr>
<td>2.3 Influences on This Work</td>
<td>32</td>
</tr>
<tr>
<td>3 Taskflow Architecture and Programming</td>
<td>35</td>
</tr>
<tr>
<td>3.1 Introduction</td>
<td>36</td>
</tr>
<tr>
<td>3.1.1 Terminology and Problem Formulation</td>
<td>36</td>
</tr>
<tr>
<td>3.1.2 Software Programming, Hardware Design, and Concurrency</td>
<td>43</td>
</tr>
<tr>
<td>3.1.3 Extending Hardware Concepts to Taskflow Programming</td>
<td>49</td>
</tr>
<tr>
<td>3.1.4 A Taskflow Environment: Key Objectives</td>
<td>51</td>
</tr>
<tr>
<td>3.2 Task Primitives, Layers, and Taskflow</td>
<td>54</td>
</tr>
<tr>
<td>3.2.1 Abstract Task Primitives</td>
<td>54</td>
</tr>
</tbody>
</table>
# 3.2.2 Layered Task Primitives .............................................. 61
# 3.2.3 Taskflow as a TaskGraph and a DataGraph .................... 64
# 3.3 Taskflow Scheduling .................................................. 65
# 3.3.1 Background: Pulse-Mode FSM ................................... 67
# 3.3.2 Task Instance Architecture ...................................... 68
# 3.3.3 Taskflow Scheduling Algorithm .................................. 75
# 3.4 XML for Taskflow Environment ..................................... 78
# 3.4.1 Taskflow Schema .................................................. 78
# 3.4.2 XML Representation ............................................... 81
# 3.5 Single-User Implementation and GUI ............................... 84
# 3.5.1 Conceptual GUI Design: A Tree View and a Graph View .... 85
# 3.5.2 GUI Implementation and Taskflow Execution Engine ....... 88
# 3.5.3 Example of Concurrent Execution ............................... 90
# 3.6 Evaluation of Taskflow Patterns ................................... 92
# 3.6.1 Synchronizing Merge ............................................ 94
# 3.6.2 Synchronizing L-out-of-M Join and Abort ..................... 96
# 3.6.3 Synchronizing Milestone ......................................... 97
# 3.6.4 Deferred Choice .................................................. 99
# 3.6.5 Recursion .......................................................... 100
# 3.6.6 Structured Cycles ................................................ 101
# 3.6.7 Other Taskflow patterns ......................................... 101
# 3.7 Summary .............................................................. 104

## 4 Web-based Collaborative Computing Environment 106

### 4.1 Introduction .......................................................... 107
### 4.1.1 Representative Project Descriptions ......................... 108
### 4.1.2 Requirements for Collaboration of Distributed Users ...... 111
### 4.1.3 Collaborative OmniDesk Environment ......................... 113

### 4.2 Asynchronous Collaboration Environment ..................... 114
### 4.2.1 Architecture of Asynchronous Collaboration ............... 116
### 4.2.2 Implementation of AGS/AGC .................................. 119
### 4.2.3 Application Examples .......................................... 121

### 4.3 Synchronous Collaboration Environment ........................ 124
### 4.3.1 Architecture for Synchronous Collaboration ................ 128
### 4.3.2 Implementation of SGS/SGC ................................... 131
### 4.3.3 Application Examples .......................................... 134

### 4.4 Web-based Environment for Tcl Applications .................. 137
### 4.4.1 Architecture for Enhancement to Safe-Tcl .................. 138
### 4.4.2 Implementation of WebWiseTcl’Tk Toolkit ................... 141
### 4.4.3 Application Examples .......................................... 150

### 4.5 Evaluation of Toolkits .............................................. 150
### 4.5.1 CollabWiseTk Toolkit ........................................... 151
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.5.2 WebWiseTclTk Toolkit</td>
<td>156</td>
</tr>
<tr>
<td>4.6 Summary</td>
<td>160</td>
</tr>
<tr>
<td>5 Taskflow Implementations of Collaborative Projects</td>
<td>161</td>
</tr>
<tr>
<td>5.1 Scalability of Taskflow Rendering and Execution</td>
<td>163</td>
</tr>
<tr>
<td>5.2 OpenDesign Taskflow Implementations</td>
<td>169</td>
</tr>
<tr>
<td>5.3 OpenExperiment Taskflow Implementations</td>
<td>177</td>
</tr>
<tr>
<td>5.4 OpenWriter Taskflow Implementation</td>
<td>186</td>
</tr>
<tr>
<td>5.5 Summary</td>
<td>192</td>
</tr>
<tr>
<td>6 Conclusions and Future Work</td>
<td>194</td>
</tr>
<tr>
<td>6.1 Summary and Conclusions</td>
<td>194</td>
</tr>
<tr>
<td>6.2 Directions for Future Work</td>
<td>196</td>
</tr>
</tbody>
</table>

Bibliography

A Definitions of a FSM and a FSMD

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.1 FSM Model</td>
<td>213</td>
</tr>
<tr>
<td>A.2 FSMD Model</td>
<td>214</td>
</tr>
</tbody>
</table>

B Specification of Cdtml Schema in XML

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
</table>

C Taskflow Programming in Cdtml

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
</table>

D Cdtml Example: Parabola Taskflow

<table>
<thead>
<tr>
<th>Subsection</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1 Main Invocation</td>
<td>230</td>
</tr>
<tr>
<td>D.2 Taskflow Definition</td>
<td>231</td>
</tr>
<tr>
<td>D.3 Taskflow Body</td>
<td>237</td>
</tr>
</tbody>
</table>
# List of Figures

1.1 WebTop: A hierarchical schematic editor and simulation environment.  
1.2 ToolWire: A commercial web-based FPGA-product design environment.  
1.3 JavaCADD: A university web-based multi-point tool environment.  
1.4 CollabTop: A collaborative environment for WebTop editor.  
1.5 Three taskflow configurations as three distinct GUI implementations.  
1.6 A taskflow configuration readily implementable in a web-browser.  
1.7 Three taskflow configurations implemented under a single GUI.  
1.8 The tree and the graph decomposition of a task \( EP \) (Evaluate/Purchase).  

2.1 Workflow architecture.  
2.2 PUNCH architecture.  
2.3 WELD architecture.  
2.4 JavaCADD architecture.  
2.5 Time space matrix of groupware applications.  
2.6 Virtual network computing's architecture.  
2.7 GroupKit architecture.  
2.8 TANGO architecture.  

3.1 A black/whitebox component definition, encapsulation and a taskflow.  
3.2 First-order ranking of programming languages for concurrency support.  
3.3 Task decomposition: a programming vs a hardware description language.  
3.4 A task encapsulation program and a trace of concurrent execution.  
3.5 Task execution views: standalone and taskflow with local or flow data.  
3.6 User-interaction view: taskflow execution of local data and flow data.  
3.7 A schema relating primitive tasks, connectors, ports, and data.  
3.8 Abstract primitives of the task model.  
3.9 Layered primitives of the task model.  
3.10 A taskflow model: an intersection of a task graph and a data graph.  
3.11 Simple 2-state FSMD models of encapsulating and scheduling a task.  
3.12 A pulse-mode FSMD model of hardware and software task encapsulation.  
3.13 The architecture of the task instance.
3.14 Summary of representative ControlJoin conditions.
3.15 FSMD specification for the architecture of an encapsulated task.
3.16 Outline of the scheduling algorithm for task $T_k$ and its successors.
3.17 A schema for XML representation of taskflow layers.
3.18 An XML schema specification for MTD.
3.19 An example for multi-task definition written in XML.
3.20 Screenshot of taskflow parabola in an XML editor.
3.21 Conceptual taskflow GUI design: a tree view and a graph view.
3.22 Implementation architecture of taskflow scheduling engine.
3.23 Concurrent execution in a taskflow: taskA, taskB, and taskC.
3.24 Summary of ten advanced workflow patterns.
3.25 Taskflow pattern: synchronized merge.
3.27 Taskflow pattern: milestone.
3.28 Taskflow pattern: deferred choice.
3.29 Taskflow pattern: recursion.
3.30 Taskflow pattern: structured cycle.
3.31 Taskflow pattern: Arbitrary cycle.

4.1 OmniDesk architecture.
4.2 Issues with task ownership and data sharing.
4.3 Shared data and shared task ownership locking.
4.4 Login session of a collaboration environment.
4.5 Three executable configurations of an asynchronous OmniFlowLite client.
4.6 Collaboration architectures of VNC vs GroupKit.
4.7 Desirable collaborative configurations for a scrollable text widget.
4.8 A high level view of client/server architecture for collaboration.
4.9 Dependency relationships for a three-participant collaborative session.
4.10 Scrollable text widget configured as shared editor and chat box.
4.11 Collaborative Tk widget demos.
4.12 Architecture for WebWiseTclTk toolkit.
4.13 New definition for command source.
4.14 Layout window of WebWiseTk toolkit.
4.15 Implementation of a toplevel window using frames.
4.16 Script to achieve a global grab on a window.
4.17 Implementation of the command menu.
4.18 Standard I/O of WebWiseTclTk toolkit.
4.19 New definition for command button.
4.20 Impress software with WebWise.
4.21 OmniFlowLite client accessible within the Web browser.
4.22 Comparison of various systems to support collaboration.
4.23 Tcl/Tk widget demos on the Web.
4.24 Evaluation of WebWiseTclTk with TkWidget demos as testbed. . . . 158

5.1 Results of conducting a series of taskflow experiments. . . . . . . . . 165
5.2 A task-data graph and its OpenDesign environment implementation. 170
5.3 OpenDesign taskflow invoking a web-based component tool at MSU. 174
5.4 Context for an encapsulated web-based component in a taskflow. . . 176
5.5 Statistical summary of an experiment with the OpenDesign taskflow. 177
5.6 Experimental design flows from two distinctive domains. . . . . . . . . 178
5.7 OpenExperiment taskflows, bringing together distributed participants. 181
5.8 Principal role models, taskflows for the OpenExperiment environment. 182
5.9 An expanded view of the taskflow configuration that implements TR30. 184
5.10 A networked permutation component encapsulated in an algorithm flow.185
5.11 Statistical summary of OpenExperiment by 5 participants on 4 hosts. 186
5.12 Client/Server Utilities for a Collaborative LaTeX Environment. . . . 187
Chapter 1

Introduction

The Internet continues to change the models of how we organize shopping, business transactions, design activities, learning, entertainment, domain-specific projects in particular, and work in general. A web-browser is a relatively simple entry-point client application to access tools and services that support a number of such tasks.

Expectations are rising to do more. Shoppers can already customize and aggregate the on-line bank and investment accounts and extensible web-browsers may remain the client applications of choice for such simple chaining of tasks. In the other extreme, industry leaders in systems-on-silicon design and biotechnology are contracting with compute farms of networked computers for customized and scalable high performance solutions, implementing a number of project-specific task sequences [1]. A survey article on network computing in the large and network computing environments provides new perspectives and directions in the rapidly evolving field [2], some of which are also addressed in this thesis.

This thesis has two major goals: (1) to create a distributed programmable and computing environment, and (2) to render such an environment collaborative. Rather than organizing distributed computing solutions in terms of point-tools and hard-wired task sequences of point-tool invocations as is currently the case, we propose an architecture to implement a programmable and computing collaborative environment where project team members can configure their computing environment in terms of the tasks for which they are responsible. As the project evolves, the task decomposi-
tions and task dependencies may change. The environment we propose will allow the participants to change it on their own, when and if the need arises.

This chapter is organized into four sections as follows:

**Background:** illustrative examples of state-of-the-art web-based tools that provide the initial context and the motivation for this research.

**Motivation:** illustrative analysis of major drawbacks of the current approach to providing hard-coded solutions in contrast to allowing the user, or a team of users working on a joint project, to program and configure the computing environment as the project evolves.

**Major goals of the thesis:** articulated by way of a representative example of a hierarchical, team-driven, taskflow tree and taskflow graph decompositions that are rendered (1) executable, and (2) collaborative across the Internet.

**Thesis organization:** outline of the chapters in this thesis.

### 1.1 Background

This project started when the World-Wide-Web was just beginning to emerge in the public view. A web-site (http://www.cbl.ncsu.edu) was organized to serve as a major depository of data sets that were evolved as *benchmarks* by the VLSI CAD workshops and conferences during 1985–95, also with the support of ACM/SIGDA for the ftp server and students at MCNC [3]. These data sets were used and continue to be used by researchers to test and report on the performance of algorithms in areas of test generation, logic synthesis and optimization, partitioning, placement and routing, etc. Concurrently, opportunities were explored to devise a methodology *and* a collaborative computing environment that could support the design and execution of distributed collaborative experiments to compare and evaluate, with statistical significance, the performance of algorithms.

Representative examples of the research in the experimental design methodology are covered in recent publications [4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16], and
MS/PhD theses [17, 18, 19]. In turn, this research influenced the direction of the work on the collaborative computing environments as presented in this thesis [20, 21, 22, 23, 24, 25, 26, 27, 28, 29]. As it turns out, since the environment is user-programmable, a number of collaborative projects other than collaborative distributed experimental design can be created and supported with this environment; a special chapter of the thesis outlines representative project examples.

For more concrete illustration of the context of this work, we introduce four examples of state-of-the-art environments that support various phases of VLSI design. Each of the environments is implemented as a web-based client. As such, these clients reveal the pros and cons of the respective implementation techniques. In principle, a combination of these techniques could be used to directly implement a specific experimental design environment – not that we propose to apply them in this manner. As we show later, our approach is radically different.

**WebTop in Figure 1.1.** This client is an entry point for a hierarchical schematic editor where user interconnects modules from a library in the chosen technology. The netlist thus created can be submitted for simulation to a number of distributed simulators. Pull-down menus are used to identify the tasks that the user is expected to select and invoke. No task chaining and scheduling is

---

![WebTop: A hierarchical schematic editor and simulation environment.](image)

*WebTop client* is an example of pre-configured design environment. Its entry point is a hierarchical schematic editor where user interconnects modules from a library. The netlist thus created can be submitted for simulation to a number of distributed simulators. This environment evolved into CollabTop in Figure 1.4 where two or more participants can collaboratively edit a schematics [30].
supported, user must select and click on each task to invoke it. This environment evolved into CollabTop in Figure 1.4 where two or more participants can collaboratively edit a schematics [30]. It is important to note that, while access to a number of tools is offered, all of the choices are ‘hard-wired’ features of the client. The nominal user has no flexibility to add another library or an alternative tool set. Since this is a university-based tool, developed at MIT, the code may be available – however, only the client’s software designer would know how to re-configure its environment.

**ToolWire in Figure 1.2.** This client represents a commercial product that has been accessible on the Web since March 2000 [31]. Several icons are displayed as buttons to be clicked when an invocation of a task is required. Pull-down menus support a number of utility tasks. The free demo version of the tool paces the user to click through a sequence of four tasks, using the icons shown in the upper-left corner: (1) upload file (in VHDL), (2) analyze file, (3) synthesize an FPGA device (as per ‘choice’ window), (4) generate a report. No task chaining and scheduling is supported, user must select and click on each task to invoke it. As in Figure 1.1, this environment is not user-configurable; it has been entirely pre-configured by the client’s software designer.

**ToolWire client** represents a commercial product, accessible on the Web since March 2000 [31]. The demo version of the tool paces the user to click through a sequence of tasks, using the four icons shown in the upper-left corner: (1) upload file (in VHDL), (2) analyze file, (3) synthesize an FPGA device (as per ‘choice’ window), (4) generate a report. As in Figure 1.1, this environment is not user-configurable.

Figure 1.2: ToolWire: A commercial web-based FPGA-product design environment.
JavaCADD in Figure 1.3. This client has been developed at MSU [32]. It has a simple-to-use front end to submit various design representations (VHDL, verilog, edif, ...) to a number of commercial tools (simulator, logic synthesizer, place & route tool, ...). Here, the user invokes, by clicking on each of the Start buttons, the two stand-alone commercial tools, (1) Synopsys logic synthesis to optimize/map a netlist to a specific library, and (2) Mentor Placement

![JavaCADD Tools](image)

![Synopsys logic synthesis](image)

JavaCADD client has been developed at MSU [32]. It has a simple-to-use front end to submit various design representations (VHDL, verilog, edif, ...) to a number of commercial tools (simulator, logic synthesizer, place & route tool, ...). Here, the user invokes, by clicking on each of the Start buttons, the two stand-alone commercial tools, (1) Synopsys logic synthesis to optimize/map a netlist to a specific library, and (2) Mentor Placement tool to place/route a mapped netlist to create rows of standard cell layout. As in Figures 1.1 and 1.2, this environment is not user-configurable.

Figure 1.3: JavaCADD: A university web-based multi-point tool environment.
tool to place/route a mapped netlist to create rows of standard cell layout. No task chaining and scheduling is supported, user must select each task to invoke it. As in Figures 1.1 and 1.2, this environment is not user-configurable; it has been entirely pre-configured by the client’s software designer. Unlike the ToolWire client, the code for JavaCADD may be available and in principle, any number of tasks can be added to its task menu. However, no task chaining and scheduling is supported, user must select and click on each task to invoke it.

**CollabTop in Figure 1.4.** This client is a collaborative hierarchical schematics editor [30], developed as an extension of WebTop (introduced in Figure 1.1). This client allows two or more users to collaboratively edit a design schematic. After editing, one of the users saves the schematics. A user can also extract a netlist from the schematics, supply test patterns, and launch a simulator. The interaction and the results of simulation can be viewed by participating users. As in Figures 1.1, 1.2 and 1.3, this environment is not user-configurable. No task chaining and scheduling is supported. Also, two or more users must negotiate the selection of each task before one of them can click to invoke it.

**CollabTop client** is a collaborative hierarchical schematics editor [30], developed as an extension of WebTop (introduced in Figure 1.1). This client allows two or more users to collaboratively edit a design schematic. After editing, one of the users saves the schematics. A user can also extract a netlist from the schematics, supply test patterns, and launch a simulator. The interaction and the results of simulation can be viewed by participating users. As in Figures 1.1, 1.2 and 1.3, this environment is not user-configurable.

Figure 1.4: CollabTop: A collaborative environment for WebTop editor.
Pros and Cons. The examples in Figures 1.1–1.4 are implemented in Java [33]. This feature makes the clients easily accessible through a web-browser. However, under this approach, output data generated by the client cannot be saved on the client’s host. If user needs to access the output data after completion of each task, he must engage in at least two more (manual) tasks: (1) visit the remote site directory where data is stored, and (2) download the output data.

There are many phases of VLSI design and subsequently, many number of tasks that must be executed and completed. Output data from one task may be required in any number of subsequent tasks. Solving design problems in an environment that has no support for task chaining is tedious and error-prone. At the level of the user-interface, tasks accessible to the user are likely major tasks that rely on chaining of tasks internally, hidden from the user. However, the software developer of the client cannot anticipate all the needs of the user, so a number of tasks are left as stand-alone tools to be invoked by the user as one-task-at-a-time. Moreover, there is no provision for the user to add to the pool of the point-tools under the common environment.

1.2 Motivation

The lack of user support for adding new tools under the same environment and the inability to schedule any chains of tasks at the level of the user-interface, exhibited by all clients in Figures 1.1–1.4, is a major drawback – and just one of the motivating factors for the research presented in this thesis.

Task pdfArchival. The purpose of this simple task is to informally introduce notions of task decomposition, task control graph chaining and scheduling, and to review the options for an intuitive graphical user-interface (GUI). Consider the task decomposed into following three subtasks, each to be completed using the tools listed:

1. download-ps (a *.ps file – using ‘Navigator’ tool)
2. translate-ps (a *.ps file to a *.pdf file – using ‘Distiller’ tool)
3. remove-ps? (remove a *.ps file, only if task (2) completes OK – using ‘rm’ tool)
We may consider three choices for GUI, all illustrated in Figure 1.5:

(a) *no task chaining*

(b) *task1/task2 chaining* (two click execution of all subtasks)

(c) *task1/task2/task3 chaining* (single click execution of all subtasks)

Any one, *but only one*, of these choices would have been implemented by the designers of the interfaces shown in Figures 1.1–1.4. In fact, when displayed, the user would either have the choice of clicking on three buttons for three tasks, on two buttons for two tasks, or one button for three task. The user with the single button interface would not have the option to decompose the task into three subtasks if and when this would be a better choice. On the other hand, a user may also get overwhelmed with MS-like interface with too many buttons and menus to consider. The task-to-task dependency edges, shown in Figures 1.5b and 1.5c, and the underlying *taskflow control graph* are not visible by the user. These edges represent the ‘hard-coded’ control flow created by the designer of the client application. Notably, the subtask *translate-ps* in
Figure 1.6: A taskflow configuration readily implementable in a web-browser.

Figure 1.5c has two outgoing edges, since the invocation of the subtask \textit{remove-ps}? is conditional.

An example where the user \textit{can} reconfigure the Netscape’s Navigator client for \textit{simple task chaining} is shown in Figure 1.6. Here the subtasks \textit{download-ps} and \textit{translate-ps} are chained, using the application configuration manager within the Navigator. A user clicking on a web site containing a \texttt{.ps} file will not only download a \texttt{.ps} file but also invoke the Distiller tool automatically to translate into a \texttt{.pdf} file. However, the task chaining in Navigator only supports unconditional task chaining, hence there is no way to invoke the \textit{remove-ps}? subtask, i.e. the conditional removal of a \texttt{.ps} file. User has no choice but to make his own determination whether or not to remove the \texttt{.ps} file.

\textbf{Task chaining under a single GUI}. One of the objectives pursued in this thesis is to provide the user with more control for task chaining and task invocation. Consider again the ‘hard-coded’ and ‘user-invisible’ task-to-task dependency edges in Figures 1.5b and 1.5c and the \textit{taskflow control graph} these figures illustrate.
Figure 1.7: Three taskflow configurations implemented under a single GUI.

A first-order solution to the problem of the taskflow control can be obtained by a simple transformation:\footnote{The simple transformation given here is a special case – it is valid for tasks with single incoming control edges only. Tasks with multiple control edges are subject to additional transformations that will be discussed in the body of the thesis.}

Let all task-to-task control-edges be visible and user-controlled, i.e. let the user set the state of each edge into one of the two possible states:

**user-enabled control-edge state** (a task driven by a single control edge can always be invoked by another task if the edge is user-enabled), and

**user-disabled control edge state** (a task driven by a single control edge can never be invoked by another task if the edge is user-disabled).

This simple transformations allows us to design a single GUI such as the one shown in Figure 1.7 that gives the user the control to click on any of the task-to-task control edge and change its state from enabled to disabled and vice versa. With a single interface, users now have the choice to visually program or configure the taskflow into any of the three pre-configured GUIs shown in Figure 1.5.
A simpler version of the interface, supporting any number of linear-only task chains, has been devised and implemented as part of an experimental class project [34], [26, 27]. The implementation presented in this thesis is comprehensive: a task-flow may be represented as a directed acyclic graph (DAG) of tasks, any task may be repeated, any task may be invoked conditionally or unconditionally, and any task may represent a hierarchical DAG itself. Tasks may be scheduled serially or concurrently, depending on the availability of hosts. The hierarchy of tasks allows the user to control the appearance of the taskflow GUI – the number of tasks the user needs to deal with at any level can thus remain manageable. A major chapter in the thesis is devoted to taskflow concepts, programming, scheduling and implementation. A companion chapter introduces techniques that render the taskflows collaborative.

1.3 Thesis Goals

This thesis has two major goals: (1) to create a distributed programmable and computing environment, and (2) to render such an environment collaborative. The background and motivation presented in the preceding sections provide the initial context to articulate these goals in more concrete terms.

Specifically, we consider a representative collaborative computing project EP (Evaluate & Purchase) that brings together a team of specialists who may well be as distributed as are the computing platforms that support this project. After introducing its goals, participants, and resources, we capture the project decomposition into tasks as a hierarchical tree as well as a hierarchical taskflow task graph, extending the GUI concepts introduced in Figure 1.7. The crux and the challenge of the proposed approach is that we create a distributed and programmable environment that allows us to capture sufficient details of the project decomposition into the respective task structures such that tasks can be scheduled for collaborative execution and completion.

Project EP (Evaluate & Purchase). The name of the project suggests an evaluation and a decision process. For consistency with the theme of earlier sections, we consider the evaluation in the context of a VLSI design phase, specifically we evaluate
the quality of the automated layout tool after the placement of cells and the routing of all connections. A number of layout tools are to be evaluated in accordance with the principles of experimental design. This implies that at least one equivalence class of cell-based netlists will be available and that each layout tool will be evaluated for each netlist instance in each equivalence class.

The project EP involves the decision to purchase the tool that receives the best overall score after evaluation of all data. The decision ‘to purchase’ is arbitrary in the context of this example. The purpose of evaluation could well have been motivated by the need to select the best layout before committing the design to manufacture, or to select the best combination of place & route algorithms, submitted for evaluation by a number of participating teams in an organized contest. For simplicity of presentation, and without loss of generality, we state the goals of the project EP as follows:

1. evaluate two layout tools;
2. submit a purchase order for the best tool.

The person expected to make the purchasing decision assumes the role of the project leader and assembles a team of participating specialists as follows:

- data supplier specialist who owns the data generator program DG;
- tool vendor specialist who owns the layout program DR1 and helpdesk program HD1;
- tool vendor specialist who owns the layout program DR2 and helpdesk program HD2;
- evaluation specialist who owns the evaluation programs DE and report summary generator program SG;
- purchase specialist who owns the purchase order program PO.

All of the project participants may be from different companies under contract with the team leader – and the challenge is to create a taskflow environment that will accomplish the goals of the project such that all programs are transparently integrated
across heterogeneous computing platforms. Before deciding on the approach, we have
to examine ways to decompose the overall goal into a number of tasks that will engage
the entire team.

**Project EP – Task Decomposition.** As per description above, the project leader
has created a list of eight available programs and five specialists that are to partici-
perate in the project. Next, he needs to organize the programs and participants into
a hierarchy of tasks. The purpose of the hierarchy is two-fold: (1) to reduce the
number of tasks that should be seen and invoked at any level of hierarchy, and (2)
to define clusters of tasks that can be invoked independently of other task clusters.
The decomposition we pursue here is clearly in terms of task and task-to-task control
edges, i.e. in terms of a taskflow task graph. We use both a tree and a graph struc-
ture to capture the hierarchical task decomposition and we argue that the tree/graph
decompositions shown in Figure 1.8 effectively maximizes the independence of task
clusters for the project *EP*.

In the tree decomposition, we see the top-level instance as the taskflow (EP), while
the programs DG, SG, and P0 are captured as task instances (DG), (SG), and (P0)
whereas the programs DR1, HD1, and DE are instances in the hierarchal instance of
(DR1--DE1). The hierarchical instance of (DR2--DE2) is composed similarly. How-
ever, it is clear also that the tree structure does not capture all dependencies between
the tasks – hence the need for the hierarchical graph as well. Only the graph struc-
ture conveys the information about the expected task sequences, e.g. that both
(DR1--DE1) and (DR2--DE2) are to be invoked by (DG), that both (DR1--DE1) and
(DR2--DE2) are subject to repeated invocations, and that (SG) synchronizes both
(DR1--DE1) and (DR2--DE2).

**A programmable, collaborative computing environment.** Elements of task
decomposition, task sequencing (including repetition), and data-dependent decision
processes are present in most any software program. Scripting languages ranging
from a UNIX shell to Tcl [35, 36, 37, 38, 39] represent a rapid ad-hoc programmable
environment to ‘interconnect’ any number of tasks, invoked as stand-alone programs.
However, this environment is not conducive for programming concurrent execution of
Project EP: goals and task decomposition.

<table>
<thead>
<tr>
<th>Goals of project EP:</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) evaluate two layout tools</td>
</tr>
<tr>
<td>(2) submit a purchase order for</td>
</tr>
<tr>
<td>the best tool</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tools mapped to tasks for project EP:</th>
</tr>
</thead>
<tbody>
<tr>
<td>$DR1$ and $DR2$ are layout tools to</td>
</tr>
<tr>
<td>be evaluated.</td>
</tr>
<tr>
<td>$HD1$ and $HD2$ are helpdesk</td>
</tr>
<tr>
<td>messengers for $DR1/DR2$ (invoked</td>
</tr>
<tr>
<td>in case of problems with $DR1/DR2$).</td>
</tr>
<tr>
<td>$DG$ compiles the netlist instances.</td>
</tr>
<tr>
<td>$DE$ evaluates the quality of each</td>
</tr>
<tr>
<td>layout.</td>
</tr>
<tr>
<td>$SG$ summarizes all evaluation</td>
</tr>
<tr>
<td>reports.</td>
</tr>
<tr>
<td>$PO$ creates the purchase order.</td>
</tr>
</tbody>
</table>

Figure 1.8: The tree and the graph decomposition of a task $EP$ (Evaluate/Purchase).

Tasks. System-level programming languages such as ADA [40] or Java [33] do provide stronger support for programming with threads. However, the effort of thread-based programming to make the environment itself as programmable and configurable as envisioned in this thesis, is non-trivial. In Section 3 we propose a simpler but an effective alternative. Programming tasks for concurrent execution is important, since we typically consider many tasks that are accessible on different hosts. In addition, an informative, user-configurable GUI is just as important.

The overriding objective is to provide a programmable environment for project
participants, not programmers. Each time, participants construct a project-specific environment capable of linking a number of programs for a limited period of time in order to meet the goals of a collaborative computing project such as EP. A project such as EP alternates between two major phases of collaboration: asynchronous and synchronizing.

During the asynchronous phase, participants work alone. The data supplier specialist who owns the data generator program DG is preparing and testing data sets. Tool vendor specialists is tuning their tools (DR1 and DR2 ) using the preliminary release of data from DG in tandem with instances of evaluation tool DE, etc. A synchronizing phase takes place only when all participants are declared ready for the automated execution of the entire taskflow. We argue that such activity does not lend itself to ‘hard-wired’ scripted solutions. Rather, a user-configurable graphical interface that can be shared and controlled by participating users is much more useful. We argue that the essential elements of such interface are the hierarchical task tree and taskflow task graph representations such as illustrated for project EP in Figure 1.8. Important components of the interface, at each level of the taskflow graph, are the user-enabled/disabled control edges introduced in Figure 1.7 in the earlier section.

A major chapter in the thesis is devoted to each aspect of the proposed environment: how to best render it functional and user-programmable and how to best render it collaborative. Highlights of implementation of the project EP are presented in a special chapter, along with an example of distributed collaborative design project, and an example of distributed of a project that involves collaborative writing, editing, and camera-ready document composition. Before closing this section, it is instructive to summarize the various notions of ‘collaborative computing’.

**What is collaborative computing?** The notion of collaborative computing continues to be applied in different contexts. In this work, collaborative computing represents a collaborative computing project, engaging a project leader and a team of distributed participants, defining and creating an environment that supports project phases such as
• project decomposition into tasks;
• assigning tasks to project participants;
• finding or creating software programs and procedures to complete each task;
• installing programs and libraries of procedures on host computers;
• deciding the sequence in which tasks are to be executed;
• choosing most appropriate data structures and protocols to protect, share, move, and archive data required by and generated by task-executing programs and procedures;
• creating a client GUI that features shared views of the project, including browsing, data editing, and execution of a task or a task sequence;
• interfacing clients to a server on a dedicated host to support all phases of the project, including asynchronous mode of collaboration where each participant works on the assigned tasks independently, and a synchronizing collaboration mode where tasks executions are being synchronized with other participants.

Groupware environments, supporting e-mail, chat rooms, video conferencing, shared white boards, are frequently equated with ‘collaborative computing’. However, we view them as facilitators of communication between collaborating participants, rather than a collaborative computing project per se. In the context of this thesis, these are point tools that can be embedded in any taskflow – if and when needed.

1.4 Thesis Outline

This thesis addresses issues in distributed collaborative computing.

Chapter 1 introduces representative examples of distributed collaborative computing that provide the scope and the motivation for the research in this thesis. The notion of a stand-alone program invoked as a task is extended to task chaining, leading to an informal representation of a taskflow as a DAG of tasks and task-to-task control edges. We show by example that a hierarchical decomposition
of a typical computational task leads naturally to such taskflows and that at each level of hierarchy, both the tree and the graph representations are useful structures for an intuitive and functional graphical user interface (GUI) – in particular when we make the task-to-task control edges user-enabled/disabled. We argue that in order to support a team engaged in collaborative computing project, we need an interactive user-configurable graphical interface that can be shared and controlled by participating users.

Chapter 2 provides an overview of earlier work related to this thesis. This includes important results from structured programming, workflow technologies, client-server computing, and collaborative computing.

Chapter 3 introduces and formalizes the taskflow architecture that allows us to create user-programmable taskflow-oriented computing environment we call *Omniflow*. The chapter consists of a number of sections, including: task primitives and the FSMD models, basic patterns of task primitives, the hierarchy of taskflow layers, the taskflow schema and the XML implementation model, the taskflow scheduling engine, and the taskflow GUI that simultaneously serves as a taskflow programming environment and the taskflow computing environment.

Chapter 4 extends the *Omniflow* architecture with a collaborative client/server architecture *OmniDesk* to support collaborative taskflow computing in two basic modes: an asynchronous and a synchronizing mode. The chapter consists of three major sections: architecture for asynchronous collaboration, architecture for synchronous collaboration, architecture to render *Omniflow/OmniDesk* environment accessible via a web-browser.

Chapter 5 includes sections of representative taskflow examples that were implemented with the *Omniflow* environment, including the project *EP* to evaluate the performance and scalability of *Omniflow* environment, a distributed VLSI design flow project, involving commercial and university-based tools, a testbed for distributed experimental design and performance evaluation of graph-based
algorithms, and a project that supports a collaborative writing, editing, and camera-ready document composition.

Chapter 6 presents conclusions and directions for future research.

Appendices A, B, C, and D include formal definitions of a FSM and a FSMD model, a full specification of XML schema for cdtML (collaborative distributed tasks markup-language in which all taskflows are captured), a brief user-guide for taskflow programming in cdtML, and a complete cdtML description of the Parabola project.
Chapter 2

Related Work

There are many ways in which to consider the problems we are addressing in this work. Contributions from both workflow management (WFMS) systems and computer supported cooperative work (CSCW) should be considered and merged whenever applicable. However, to manage the scope of our effort, we consider a number of project-centric views that lead us to task-specific projects that typically involve relatively small number teams. We argue for and propose an approach that matches well with the project model we consider.

Most of the existing systems can be classified under one of the following categories:

- a workflow management system that provides access to distributed tools and data over the network for a very specific application, or
- a collaborative environment for sharing applications among distributed users.

We next describe in detail several existing systems in both the domains and also discuss their limitations in the context of our work.

2.1 Workflow Technologies

Several workflow management systems (WFMS) have evolved over the years, each catering to a wide variety of applications. Therefore, the concept of workflow management can imply a different meaning depending on its application areas such as
in the consulting area, the research domain, or the industrial sector. One such definition, given in [41], describes a workflow management system as an active system that manages the flow of business processes performed by multiple persons and gets the right data to the right people with the right tools at the right time. Most of the workflow management systems have focused greatly on traditional business environments and enterprise computing that merely access database systems. The need for workflows in more sophisticated scientific problem-solving environments, dubbed as scientific workflows, have been rightly emphasized and discussed in [42].

2.1.1 WorldFlow - Virtual Warehouses using Workflows and the Web

WorldFlow [43, 44] applies the technology developed for workflows to meet the needs of electronic commerce applications over the Web. It has been developed in Krithi Ramamritham’s Database Systems Laboratory at University of Massachusetts.

Traditional workflow management systems are provided with additional information about the Web, its protocols and data format, such that workflows can directly access information by interacting with Web servers. This allows companies to automate their business processes on the Web by allowing steps to automatically query/update information from Web servers located both within and outside the organization. The additional knowledge required is provided by the workflow system developer and the workflow designer. The main components of workflow management over the Web are shown in Figure 2.1.

A workflow application called StockAdvisor has been designed and implemented. The application is a tool for making "well-informed" stock investment decisions and helps automate the process of bringing the necessary data to provide the specific information needed by a user.

Limitations. Since the traditional workflow management system is enhanced to work for the http protocol, it can only access database systems via the Web-server or agents. The environment is primarily designed for performing business transactions over the Web and is not suitable for tasks that are typically performed in any generic
project. It also lacks support for synchronous shared environment.

2.1.2 PUNCH - Purdue University Network Computing Hubs

PUNCH [45, 46] is an infrastructure for network-based VLSI CAD and TCAD that allows users to access and run existing software tools via standard world-wide web browsers.

Functionally, PUNCH allows users to: (a) upload and manipulate input-files, (b) run programs, and (c) view and download output - all via standard web browsers. Its infrastructure is divided into two parts, as shown in Figure 2.2. The front-end primarily deals with data management and user-interface issues. The "hub-engine" serves as PUNCH’s user-transparent middleware. It consists of a collection of hierarchically distributed servers that co-operate to provide on-demand network-computing. This part of the infrastructure addresses the following issues: management of the run-time environment, security, control of resource access and visibility, and demand-based scheduling of available resources.
The earliest implementation of PUNCH has been operational since April 1995. The current hubs contain over thirty tools from eight universities and four vendors, and serve more than 500 users. It currently provides these services for tools with text-based and graphical user-interfaces, limited to X-based tools.

Limitations. Users are limited to accessing resources only available within PUNCH and cannot create or configure workflows that access other tools over the network. It currently supports only the Web-based http protocol. The PUNCH system lacks support for shared environment and floor control for real-time synchronous collaboration.

2.1.3 WELD - An Environment for Web-based Electronic Design

WELD [47, 48] aims to provide a reliable, scalable connection and communication mechanisms for distributed users, tools and services. It relies on a new set of semantics
referred to as BASE (Basically Available, Soft state, Eventual consistency) that stresses availability and low latency, instead of traditional transactional capabilities of ACID. In BASE, a fast, approximate answer may be more valuable initially than a late-but-correct response. WELD tries to strike a balance between the two properties.

![WELD Architecture Diagram](image)

Figure 2.3: WELD architecture.

The three-tier WELD architecture, shown in Figure 2.3, consists of:

- **Remote servers** provide access to either command-line tools encapsulated by server wrappers or tools with built-in support for socket connections and WELD communication protocols.

- **Network services** incorporate infrastructure components on demand, such as distributed data manager, proxies, and registry service.

- **Clients** applications use the WELD infrastructure to access network resources. Clients are either Java browser clients, or generic clients, developed in socket-enabled languages such as C, C++, perl, etc using WELD protocols.
Limitations. Although command-line tools can be easily encapsulated on the remote server by using wrappers, other tools need to be re-written to conform to WELD communication protocols. In addition, tools which do not have built-in support for socket connections cannot be encapsulated on the remote server. Addition of new network protocols is not convenient and hence access to tools using such protocols is limited. Support for collaboration is also limited - it does not provide synchronous shared environment.

2.1.4 WorkXpert - A Process Management Tool

WorkXpert [49] is an integrated suite of Workflow/ Technical Process Management tools used to capture, manage, and track design processes for project teams. WorkXpert is a commercial product developed by Mentor Graphics [50] and consists of three products:

1. **FlowXpert** is a runtime, multi-user process manager for designers. It manages process execution, monitors and controls process dependencies and provides point and click access to tools and utilities. It also provides graphical views of the processes, a report builder, shared whiteboard, and attachment of annotations.

2. **ProjectXpert** is a tool for project managers to track key project deliverables, view the project status, import/export project schedules, etc.

3. **XpertBuilder** is a tool for capturing and building workflows.

Limitations. It provides an environment over a local area network only and cannot access tools using remote services. It does not even support access to tools using Web interfaces. For synchronous collaboration, it has provision for few simple tools such as shared whiteboard.
2.1.5 JavaCADD - Java-based Distributed ECAD Services

JavaCADD [32, 51] is a Java-based server and client GUI used for providing distributed ECAD services. The server/client combination allows user access to batch-oriented ECAD services, such as synthesis, place/route, etc., without having to provide full login-access for the users.

![JavaCADD architecture](image)

Figure 2.4: JavaCADD architecture.

The architecture for the JavaCADD system is shown in Figure 2.4. The JavaCADD client is generic and on initialization, queries the server to retrieve the list of services/tools available at the server. It then dynamically creates a GUI for the requested tool service on the client, based on the templates stored at the server for GUI definition. Once the user specifies the design data and the parameters, the JavaCADD client submits those to the server for invoking the tool, and upon completion, retrieves the results to the client side transparently. Java Remote Method Invocation is used for client/server communication. The architecture supports addition of new services on the server side, without having to update the client software.

Both client and server are Java applications, and share the operating system inde-
pendence. JavaCADD has been used by undergraduate classes at MSU to access the ECAD applications and computing resources located at the MSU/NSF Engineering Research Center.

**Limitations.** Remote services are limited to access to tools encapsulated on the server for distribution. Arbitrary sequencing of more than one service by the users is not feasible. This environment also lacks support for shared environment.

### 2.1.6 FlowMake - A Workflow Modeling and Verification Tool

FlowMake [52, 53] provides workflow analysts and designers a well-defined framework to model and reason about various aspects of workflows. It introduces four modeling objects:

1. a *task* represents the work to be done to achieve some given objectives;

2. a *condition* is applied to represent alternative paths in workflow specification depending on a conditional value that is dependent on external parameters;

3. a *synchronizer* waits for the completion of more than one execution paths before proceeding further; and

4. a *flow*, represented by a directed arrow, defines the connection between any two objects.

Large workflow graphs, created using these four objects, are prone to error situations. A set of constraints ensure correctness of workflow specifications. The verification engine developed is used to check the consistency of the workflow models.

**Limitations.** The four modeling objects may be well-suited for describing business workflows, but are insufficient to describe the heterogeneous environment of a typical project. The FlowMake prototype has been implemented under Microsoft Windows NT/95 platform using visual C++ and is not available on other platforms.
2.2 Computer Supported Cooperative Work

Computer supported cooperative work (CSCW) and groupware systems deal with collaboration necessary to engage a group of people to achieve a common task or goal and provides an interface to a shared environment. One of the classification schemes for the groupware systems is based on time space matrix [41], and is shown in Figure 2.5.

- **Same place, same time**: The most popular and conventional example is face-to-face interaction where several people sit together at the same time at the same place and perform a group activity.

- **Same place, different time**: Multi-user editor is an example which allows asynchronous editing of a document by several people at different times.

- **Different place, same time**: Shared white-boards and chat tools are examples of conference systems that enable people distributed among various places to hold synchronous distributed meetings.

- **Different place, different time**: Newsgroups and e-mail are a good example of asynchronous distributed interaction where people read and post messages to a newsgroup at different times from a different place or use e-mail to interact with one another.

Systems that attempt to support the latter two, synchronous as well as asynchronous distributed interactions, are the most interesting and is an area of continued research.

<table>
<thead>
<tr>
<th></th>
<th>Same time</th>
<th>Different time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same place</td>
<td>face-to-face</td>
<td>asynchronous</td>
</tr>
<tr>
<td></td>
<td>interaction</td>
<td>interaction</td>
</tr>
<tr>
<td>Different place</td>
<td>synchronous</td>
<td>asynchronous</td>
</tr>
<tr>
<td></td>
<td>distributed</td>
<td>distributed</td>
</tr>
<tr>
<td></td>
<td>interaction</td>
<td>interaction</td>
</tr>
</tbody>
</table>

Figure 2.5: Time space matrix of groupware applications.
2.2.1 VNC - A Virtual Network Computing Environment

Virtual network computing (VNC) [54, 55] has been developed by the Olivetti Research Laboratory, which is now part of AT&T Laboratories, Cambridge. It is a very thin client for remote display system but highly portable due to the simplicity of the protocol used. It is available on a variety of platforms including solaris, linux, windows and macintosh.

VNC is, in essence, a remote display system which allows one to view a computing ‘desktop’ environment not only on the machine where it is running, but from anywhere on the Internet and from a wide variety of machine architectures. It uses a simple protocol, whereby a server scans the pixel of the displayed image in the frame buffer and asks the remote client to display the same pixel on its screen. Also the pixels are transmitted in a highly compressed format and the updates are sent only when the client explicitly requests for an update. This results in a highly efficient implementation which works reasonably fast even over a slow connection. Collaboration is achieved because the server allows multiple users to connect to the same display screen. Figure 2.6 shows the basic architecture of VNC for collaboration.

![Centralized VNC Server](image)

Figure 2.6: Virtual network computing’s architecture.

**Limitations.** VNC basically uses a centralized architecture and results in performance degradation when number of participants increase. It also does not provide good support for floor-control and group awareness.
2.2.2 GroupKit - A Groupware Applications Development Toolkit

GroupKit [56, 57, 58] is an extension toolkit to Tcl/Tk that allows the development of groupware applications such as drawing tools, editors, and meeting tools for sharing among several users simultaneously. GroupKit has been developed at University of Calgary. A commercial company called TeamWave Workplace [59], which provides an Internet forum for teams to coordinate, collaborate and share information, is a marketing product based on GroupKit.

![GroupKit Architecture Diagram]

Figure 2.7: GroupKit architecture.

GroupKit is mostly a replicated run-time infrastructure. It actively manages the creation, location, interconnection, and teardown of distributed process; communication setup, such as socket handling and multi-casting; and groupware specific features such as providing the infrastructure for session management and persistent conferences.

The run-time architecture of a typical groupware application, using GroupKit, is
shown in Figure 2.7. The registrar is a centralized process that acts as connection point for a community of conference users. GroupKit programmers build both: (a) session managers which provides user interface and a policy as to how users enter or leave conferences, etc, and (b) conference applications (e.g. shared editor, whiteboard, etc.) which are invoked by the user using session manager.

Limitations. GroupKit is a toolkit for creating collaboration-aware applications using Tcl. It cannot make single-user applications collaborative without modifying the source code.

2.2.3 CBE - A Collaboratory Builder’s Environment

Collaborative Builder’s Environment [60, 61] is a toolkit written entirely in Java that provides a framework, DistView, for building collaborative applications. DistView has been developed at University of Michigan. It provides group communication services that meets the various shared state management needs of collaborative environments. It provides a rich interface for group and session management, the ability to ensure totally ordered message delivery, a lock-based distributed synchronization mechanism, and support for selective window sharing.

The DistView framework has been implemented and used in context of UARC [62] project, an experimental testbed for wide-area scientific collaborative work. Various facilities provided to users include shared data viewers for graphically displaying instrument data, a multi-party chat box, and a drawing tool for sharing notes, images and drawings.

Limitations. CBE’s DistView is a framework for creating collaboration-aware applications using Java. It basically uses a centralized architecture and may result in performance degradation under certain circumstances. It cannot render single-user applications collaborative without modifying the source code.
2.2.4 TANGO - A Collaborative Environment for the World-Wide Web

TANGO [63, 64] is an integration platform to enable implementation of multi-user Web-based collaborative environments. The main functionality, provided by the system, consists of the following elements:

1. *Session management:* It supports (a) creating a session, (b) joining an existing session, and (c) leaving a session. Every session has only one master who has the special privileges of controlling the application behavior and/or controlling access to other users of the session.

2. *Communication:* Two types of messages can be sent by an application to its peers: (a) control messages for communication with the server, daemons, etc., and (b) application messages for communication between user application.

3. *User authentication:* Only authorized users are allowed to access and participate in TANGO sessions.

4. *Event logging:* All communication messages are recorded in a database.

The architecture of the TANGO system is shown in Figure 2.8. Each client runs a local daemon that communicates with the main central server for collaboration and manages local applications. The daemon, implemented as plug-in to Web browsers, is the operating system dependent core part of TANGO.

TANGO has an API which allows a developer to port a single-user application into a shared application for collaboration. A developer must define: (a) when to share an event, (b) what to share as an event, and (c) what to do after an event is received; thereby creating a collaboration-aware application. The current applications, based on TANGO, have been developed in areas such as health care [65] and distance education [66].

**Limitations.** The environment does not support workflow based access to distributed tools and data. It is merely an environment to share applications for col-
laboration. Developers need to identify and define each event for sharing when porting single-user applications to the TANGO system. There is no facility for sharing collaboration-unaware applications.

2.3 Influences on This Work

We have discussed a few workflow management systems here that are closely related to the problem. A comparison of several other workflow technologies using its metamodels is found in [67]. Distributed programming technology like CORBA [68] makes multiple servers transparent from users, but require the applications to be CORBA-compliant.

In the context of this thesis, our work has evolved over several years with implementations of early prototypes found in the Reuben environment [21, 22] and the OmniFlowLite environment [26, 28]. Our research on taskflow programming basically supports the fundamentals of structured programming described in [69, 70, 71, 72] and
essentially makes use of blackbox and whitebox software components [73, 74, 75, 76]. The two software components form the basis of the taskflow architecture where the blackbox component is used to represent a wide variety of tools, ranging from legacy applications to newly written programs, and the whitebox component is used to provide a highly interactive GUI-based taskflow design and execution environment.

Rather than relying on the formalisms of alternative approaches such as Petri nets [77, 78], Actor Computations [79, 80, 81], Action Systems [82], etc., we have chosen the simpler hardware-based FSMD model [83, 84, 85] to represent, encapsulate and schedule each blackbox as well as whitebox component for execution. Moreover, we have adopted and adapted a scripting language, Tcl [39], and approaches rooted in hardware description languages (HDL) such as Verilog [86], VHDL [87], or HardWareC [88] to formalize and support the concepts of concurrency at a much higher level of abstraction than found with threaded programming in system level languages [33, 40, 73, 89].

We have also discussed two classes of groupware systems in this thesis. Groupware systems [90] either have (1) a centralized architecture where a single application program, residing on one central server machine, controls all input and output to the distributed participants; or (2) a replicated architecture where an application program, installed and running at each site, coordinates explicitly both local and remote actions and also maintain synchronization with each other. We proposed and implemented an architecture that uses a hybrid approach, taking advantage of both approaches while avoiding their respective limitations.

A mathematical model of groupware systems called Team Automata is presented in [91]. Team Automata are mathematical compositions of a set of Component Automata which interact via shared actions and can have an infinite number of states. The model classifies actions, which take an automata from one state to another, into three types: (1) input actions, (2) output actions, and (3) internal actions. For shared action, one or more automata specify, within their input action sets, the same action as one or more other automata specify within their output action sets. Such a model is useful to reason about the correctness and performance of groupware systems. This model played an initial role in design of our hybrid architecture where
we configure each widget into two states: **Interact** and **Observe**. The output actions above result when in interact state whereas the input actions result in observe state. Additionally, the application examples such as the shared calendar, called **The Electric Secretary** [37], and the shared whiteboard [92] played an important role in the tk-based implementation of asynchronous and synchronous group client/server architectures.
Chapter 3

Taskflow Architecture and Programming

This chapter introduces and formalizes the taskflow architecture that allows the user to create a programmable taskflow-oriented computing environment we call Omni-Flow. In this environment, user can create a highly interactive composition of well-defined distributed components, each of which may be executable on its own host. The interactivity allows the user to isolate groups of components from each other or to schedule the execution of several tasks by each of the component, serially or concurrently, depending on the availability of the host.

This chapter is organized into following sections as follows:

*Introduction*: terminology that defines components, component encapsulation and task and taskflow instances in terms that are specific to the work in this thesis. In particular, this section finds important metaphors in component-based VLSI design that have a direct bearing on the approach to capturing and scheduling the execution of the taskflow.

*Taskflow primitives*: a formal introduction of abstract task primitives, layered task primitives, and the taskflow model of its control and data graphs.

*Taskflow scheduling*: an introduction to pulse-mode FSMs, the encapsulated component architecture, and the taskflow scheduling algorithm.
Taskflow schema: an outline of the XML representation of a taskflow schema.

Taskflow GUI implementation: highlights and illustrations of taskflow GUI implementation.

Taskflow execution: an analysis of representative taskflow patterns and synchronization, example and analysis of the taskflow concurrent execution.

3.1 Introduction

This section introduces terminology and basic concepts that are to be developed in the body of this chapter. We point out the lack of consensus in the literature about what is and what is not a software component, an object, an encapsulation, and an instance. Hence, we are particularly careful to define the notions of a blackbox and whitebox software component, its encapsulation and its instantiation. In doing so, we also review the relationship to hardware description languages and methodology that is to play an important role in our approach to capturing software component instances, their interconnections, and their scheduling as taskflows. In the process, we develop a point of view that guides our approach to define a set of objectives for a taskflow-oriented programming and execution environment that is to be formalized and implemented in the sections that follow.

3.1.1 Terminology and Problem Formulation

In presenting the work of this thesis, we need to borrow terminology from areas of software engineering, workflow technology, CSCW (computer-supported collaborative work), as well as the design of electronic hardware components and systems. While the notion of a component in a hardware technology would not ordinarily require much elaboration, this is no simple matter in software technology. In [73], we find a complete book chapter, entitled What a component is and is not, that provides an authoritative analysis on the subject. We quote from the introduction of the chapter:

The terms ‘component’ and ‘object’ are often used interchangeably. In addition, constructions such as ‘component object’ are used. Objects are said to be
instances of classes or clones of prototype objects. Objects and components are both making their services available through _interfaces_, and interfaces are of certain _types_ or _categories_. As if that was not enough, object and component interactions are described using object and component _patterns_ and prescribed using object and component _frameworks_. Both components and frameworks are said to be _whitebox_ or _blackbox_, and some have even shades of _gray_ and _glassboxes_. Language designers add further irritation by also talking about _namespaces, modules, packages_, and so on.

This plethora of terms and concepts needs to be either reduced by eliminating redundancies or unfolded, explained, and justified .....

The chapter emerges with a formal definition of a software component, as one of the outcomes of the Workshop on the Component-Oriented Programming [74]:

‘A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.’

Nine more definitions of software components, ranging from Booch [75] to Sametinger [93] are summarized in a survey chapter _What others say_ [73]. In the context of our work, the component definition offered by Wiederhold, Wegner, and Ceri in [76] is closest to the motivation and the approach in this thesis:

‘Megamodules are internally homogeneous, independently maintained software systems [...] Each megamodule describes its eternally accessible data structures and operations and has an internally consistent behavior.’

Be that as it may, components are for composition – and a compatible re-arrangement and re-use can lead to new, more powerful, software compositions. Components that must be compliant with protocols such as _RPC, CORBA, JavaRMI, SOAP_ [68, 94, 95, 96, 97] may not mix and rearrange readily. In this thesis, we consider components that do not rely on these protocols. Specifically, examples of components that we consider are truly _blackbox_ components: any legacy program on a host Z, accessible with a telnet or ssh client, any CGI-script program, accessible on a web-server with a http client, any program hosted by an application-specific server, accessible with a secure socket-based client, etc. As such, these components fit the description of a ‘software component as a unit of composition’ formalized above and in [74, 76]. For each of _blackbox_ component we know only the following:
1. command line which invokes the component for execution;
2. description of input data to be read by the component;
3. description of output data to be written by the component;
4. information on how to ascertain the completion status of the component. In the simplest case, we may compare the timestamps of the input and output data.

In general, to present the work in this thesis in the simplest possible terms, we shall introduce definitions and notation that may be different from the ones used elsewhere in the literature, though the words may be the same. In particular, the notions of the blackbox and whitebox components and their encapsulation, a task and a taskflow instance, are specific to the approach we propose.

**Blackbox component (Definition).** A blackbox component (BBC) \( k \) is a stand-alone program executable on a specific host. It is represented as a box with several ports: an invocation control port, a status control port, any number of input data ports, and any number of output data ports. When invoked and executing, it may read input data sets \( \mathcal{D}_{I_k} \), it may write output data sets \( \mathcal{D}_{O_k} \), and it is expected to terminate and signify completion. For some blackbox components, we may deduce the completion status by comparing the timestamps of the input and output data sets.

An encapsulated blackbox component (eBBC) represents a finite-state-machine (FSM) arrangement with a blackbox component, where the blackbox component is simply an extension of the data path, communicating with the FSM by way of two handshaking signals. Such arrangements are common in the high-level synthesis and the design of hardware systems [83, 84, 85]. Specifically, a formal definition of a FSM (finite-state-machine) and a FSMD (finite-state-machine with a data path) is given in [83] and also described in Appendix A. A signal to invoke the blackbox is issued by the invoking FSM, which in turn is invoked by the user or another program. The illustration of a blackbox component and its encapsulation is shown in Figure 3.1. More details about the signaling will be given later.
A blackbox component (BBC) is a stand-alone program executable on a specific host. It has an invocation control port, a status control port, any number of input data ports, and any number of output data ports. An encapsulated blackbox component (EBBC) represents finite-state-machine (FSM) arrangement with a datapath, where the blackbox component is simply an extension of the data path, communicating with the FSM by way of a two handshaking signals. Such arrangements are common in high-level synthesis and design of hardware systems [83, 84]. A blackbox task instance (or simply a task instance) is created after we (1) encapsulate the blackbox and (2) attach data to it.

A whitebox component is the composition of two or more task instances or taskflow instances and decision boxes. The decision boxes are required to synchronize the execution of task or taskflow instances. Later in the thesis we re-define task instances so they contain decision box elements themselves. An encapsulated whitebox component represents a FSM arrangement with adatapath – a taskflow.

Figure 3.1: A black/whitebox component definition, encapsulation and a taskflow.

A blackbox task instance (or simply a task instance) is created after we (1) encapsulate the blackbox and (2) attach data to it. The finite-state-machine is configured with task specific information once we create a task instance. The process of invoking and executing any encapsulated blackbox component is denoted as a task execution.
**Whitebox component (Definition).** A *whitebox component* (WBC) $q$ is the composition of two or more *task instances* or *taskflow instances*\(^1\), and *decision boxes*. The decision boxes are required to synchronize the execution of task or taskflow instances. Later in the thesis we re-define task instances so they contain decision box elements themselves, and the whitebox itself becomes the composition of two or more task or taskflow instances only. Just like a blackbox, the whitebox is also represented as a box with several ports: an *invocation control port*, a *status control port*, any number of *input data ports*, and any number of *output data ports*. When invoked and executing, it may read input data sets $D_i$, it may write output data sets $D_o$, and it is expected to terminate execution and signify completion.

Similar to encapsulated blackbox component, an *encapsulated whitebox component* (eWBC) represents a finite-state-machine (FSM) arrangement with a datapath, where the whitebox component is simply an extension of the data path, communicating with the FSM by way of a two *handshaking signals*. A signal to invoke the whitebox is issued by the invoking FSM, which in turn is invoked by the user or another program. We denote the *encapsulated* whitebox component as a *taskflow*. The illustration of a whitebox component and its encapsulation is shown in Figure 3.1.

A *whitebox task instance* (or simply a taskflow instance) is created after we (1) *encapsulate* the whitebox and (2) attach data to it. The finite-state-machine is configured with task specific information once we create a taskflow instance. The process of invoking and executing any *encapsulated* whitebox component is denoted as a *hierarchical task or taskflow execution*. A whitebox component is clearly hierarchical and can contain instances of other whitebox components.

**Taskflow instance (Example).** We now review the taskflow instance in Figure 3.1 again. The purpose of the example is to illustrate a plausible scenario where we want to create, with a minimum of effort, a new program component from existing components. As shown, there are three task instances, labeled as solverOld(1), solver(1), and solver(2). Solver(1) and solver(2) are instances of the same component. The need for the decision boxes labeled as DL1, DL2, DL3 becomes clear once we analyze

\(^1\) The definition is recursive and the taskflow instance is defined within.
the expected behavior of the interconnected components and the objectives of this taskflow. There are two types of edges in this graph: control edges (bold) and data edges (thin). Starting with the invocation signal from the FSMD, the decision box DL1 may invoke one or both of the successor tasks, depending on the data values from \( D_m \). After invocation, each task instance generates the control signal about the task completion that is again interpreted, in the successor decision box, possibly with data, before making the decision which task to invoke next. Here is the short description of the taskflow and its objectives:

- \( \text{solverOld}(1) \) is an instance of a legacy program, known to find acceptable solutions even for very large problem instances.

- \( \text{solver}(1) \) is an instance of a whitebox program prototype, expected to find better solutions than \( \text{solverOld}(1) \) at least for smaller problem instances.

- \( \text{solver}(2) \) is another instance of the whitebox program that may be reinvoked after the completion of either \( \text{solverOld}(1) \) or \( \text{solver}(1) \), using the best of the solutions from these two programs as its starting point.

- When the size of the problem instance \( D_m \) is ‘large’, the decision box DL1 selects only \( \text{solverOld}(1) \) for execution, execution of \( \text{solver}(1) \) is skipped, and so is the execution of \( \text{solver}(2) \).

- When the size of the problem instance \( D_m \) is ‘medium’, the decision box DL1 selects both \( \text{solverOld}(1) \) and \( \text{solver}(1) \) for concurrent execution. Decision box DL2 evaluates the quality of results from both solvers. If \( \text{solverOld}(1) \) returned better solution than \( \text{solver}(1) \), \( \text{solver}(2) \) is invoked with the solution from \( \text{solverOld}(1) \) as the starting point, otherwise, the execution of \( \text{solver}(2) \) is skipped.

This simple example illustrates a number of principles about component encapsulation, instantiation, and scheduling. We shall revisit the example later in the thesis when we discuss these concepts in more detail.
**Problem formulation.** We are now ready to formulate an abstract view of the problem we are solving in this thesis:

| Given: | \( L \) blackbox (executable) components and data  
| \( M \) whitebox (executable) components and data  
| \( N \) owners \((N \leq L + M)\)  
| \( K \) hosts \((K \leq L + M)\) |

| Problem: | create a new (and executable) whitebox component, i.e. program a taskflow with up to \((L + M)\) components. |

We shall rely on a number of techniques to implement a simple to use environment to create and execute large taskflows. In choosing the FSM model to encapsulate each blackbox as well as each whitebox component, we have shown preference for traditional (and simpler) hardware-based solutions over alternative approaches that may rely on the formalisms of Petri nets [77, 78], Actor Computations [79, 80, 81], Action Systems [82], etc. Electronic circuit design in particular has a long tradition of addressing problems of concurrency and synchronization. The design of interacting FSMs, synchronous and asynchronous, is the norm. In fact, judging from an advance program of the 2001 conference, the design of asynchronous VLSI circuits is alive and well [98].

On the other hand, the approach we propose also readily supports the basic tenets of structured programming [69, 70, 71, 72]:

- creation of simple task sequences;
- data-dependent decisions for a block of task sequences;
- a data-dependent iteration for a block of task sequences;
- component encapsulation for a block of task sequences;
- single entry and single program exit point for each encapsulated component.

All features of this programming style except iteration are supported in the taskflow example in Figure 3.1. Examples with iteration where tasks are repeated, and
the number of iteration depends on data, are shown later. With the FSM-based encapsulation model as proposed here, the implementation is relatively simple.

### 3.1.2 Software Programming, Hardware Design, and Concurrency

Large software systems may contain on the order of one million lines of code. A single integrated circuit representing a large hardware system today may contain more than a million transistors. Traditionally, the approach to designing a large software system differs from the approach to designing a large hardware system. Designers of hardware systems rely on complex software systems to analyze, synthesize, and simulate models of the hardware system and its components before and after these components are realized at the mask level and committed to fabrication. We find a number of hardware design techniques highly relevant when exploring approaches to designing the whitebox environment. Specifically, we consider several hardware design techniques:

1. hierarchical decomposition into tasks, where each task is bound to a specific hardware component;
2. explicit design of the control flow to sequence all tasks, maximizing the concurrency under assumption of no constraints on the available hardware resources;
3. explicit typing of all I/O variables on the boundary of each component;
4. explicit control to test and re-test the tasks performed by each component in the standalone mode even after embedding the component in the overall system.

While hierarchy is an accepted technique when managing software projects, the notion of maximizing the concurrent execution of tasks is much harder to apply and implement in a software design project than it is in a hardware design project. The technique of explicit typing of I/O variables on the component boundary is mandatory in VLSI hardware design, and the technique of testing each component independently
before and after its embedding is an accepted practice when delivering systems that are to be close to 100% free of manufacturing defects.

In terms of programming effort, scripting languages [38] are designed for "glueing" applications (i.e. components); they use typeless approaches to achieve a higher level of programming and more rapid application development than system programming languages. However, scripting languages are still limited to sequential execution of application programs on a single computer. Concurrent execution of application programs, if any, is either handled by adhoc programming constructs in scripts or is simply left to the application program for implementation. A first-order estimate of the programming effort vs. the level of concurrency support is illustrated in Figure 3.2. Given limited time and resources, using system languages such as Java [33], ADA [40] or Component Pascal [73, 99] has not been an option in this project.

![Diagram of concurrency support and programming effort](image)

Figure 3.2: First-order ranking of programming languages for concurrency support.

Rather, we adopted and adapted a scripting language (Tcl) and approaches rooted in hardware description languages (HDL) such as Verilog [86], VHDL [87], or HardWareC [88]. In addition to (1) formalizing the component encapsulation process introduced in the preceding section, we also devise and implement (2) an XML schema that captures the interconnection of encapsulated components, (3) a scheduling algorithm that executes the components concurrently whenever there are available hosts, (4) a graphical user interface that support high level of user interactivity with the scheduled component execution. Conservatively, as shown in Figure 3.2, we rank the
programming effort at the level of a scripting language, but much higher level of concurrency. Future experiments will decide with more precision the true gains of the proposed approach.

Two simple examples in Figures 3.3, 3.4 motivate the approach and serve to illustrate issues that are explored more fully in the sections that follow.

**Task decomposition for maximum concurrency**

Figure 3.3 depicts an example where the objective is to compute an output $C3$ in terms of the input $A1$. We provide two descriptions side by side: one in a programming language Tcl [39], the other in a hardware description language verilog [86]. On the first reading, the 4-task decomposition can be most readily identified in the Tcl description:

- **Task $T1$** reads data $A1$, performs a number of operations on the input data, outputting new data as $B1$ and $C1$;

- **Task $T2a$** reads data $B1$, performs a number of operations on the input data, outputting new data as $B2$;

- **Task $T2b$** reads data $C1$, performs a number of operations on the input data, outputting new data as $C2$;

- **Task $T3$** reads data $B2$ and $C2$, performs a number of operations on the input data, outputting new data as $C3$.

Tasks listed in the Tcl description are implemented as Tcl procedures. The schedule of task executions is implicit in the description; all tasks are invoked serially, in the order shown. Tasks $T2a$ and $T2b$ invoke the same procedure $P2$, each time with different input data. Since $C3$ does not depend in which order $T2a$ and $T2b$ are executed, both tasks could be invoked concurrently. A concurrent execution of $T2a$ and $T2b$ could speed up the overall task execution significantly.

Consider now the verilog description in Figure 3.3. The verilog description represents a netlist of five components, labeled as *cells*. Instance of cell1 implements task
Major objective of this task decomposition: for input data $A_1$, generate output data $C^3$

```tcl
# a tcl
# description
# =========
proc X {A1 _C3} {  
    upvar _$C3

    # task T1
    P1 $A1 B1 C1

    # tasks T2a, T2b
    P2 $B1 B2
    P2 $C1 C2

    # task T3
    P3 $B2 $C2 C3
}
```

```verilog
// a verilog description
// ================
module X (modIn, modOut);
input [1:0] modIn;
output [1:0] modOut;
wire i_t1, A1, t3b_o, C3;
assign {i_t1, A1} = modIn[1:0];
assign modOut[1:0] = {t3b_o, C3};
wires

cell1 T1 (.i1(i_t1), .i2(A1), .o1(t1t2a), o2(t1t2b),
    .o3(B1), o4(C1);
cell2 T2a (.i1(t1t2a), .i2(B1), .o1(t2at3a), .o2(B2);
cell2 T2b (.i1(t1t2b), .i2(C1), .o1(t2bt3a), .o2(C2);
cell3 T3a (.i1(t2at3a), .i2(t2bt3a), .o1(t3at3b));
cell4 T3b (.i1(t3at3b), .i2(B2), .i3(B3), .o1(t3b_o),
    .o2(C3);
endmodule
```

The simplest approach to schedule tasks in a programming language is to invoke them serially; see the four tasks in the tcl module.

Given unlimited hardware components, the simplest approach to schedule tasks in a hardware description language is to maximize concurrency. Here, data and control I/O signals are explicitly defined. A synchronizing task $T3a$ has been added.

Figure 3.3: Task decomposition: a programming vs a hardware description language.
T1, two instances of cell2 implement tasks T2a and T2b, the combined instances of cell3 and cell4 implement the task T3 as a chain of two tasks: T3a and T3b. The task T3a performs synchronization of tasks T2a and T2b and will be explained shortly. A netlist in this format can be read by a logic-level simulator, a logic optimizer, or a place-and-route tool. In each case, a component or cell library is also read, with the definition and relevant implementation details about each component or cell. The logic-level simulator analyzes the design behavior and may also detect presence of hazards and races for certain sets of inputs. The logic optimizer may restructure the netlist into one with less components, less delay, or both. The place-and-route tool uses the netlist to physically place components on a plane and route the interconnections between the components to minimize the connection length and the overall layout area.

The relative simplicity of expressing the intrinsic concurrency of the 4-task decomposition, captured by the verilog description in Figure 3.3, is instructive. This is clearly a direct implementation of maximum concurrency – it relies on the freedom to choose two instances of cell2 rather than implementing tasks T2a and T2b serially with a single instance of cell2 (a common practice in hardware design when trading device speed versus device area). All control signals to sequence the task invocations are an intrinsic part of the hardware implementation: initializing $i_{-}t1 = 1$ invokes task T1 which in turn invokes Tasks T2a and T2b once the output control signals generated by T1 is set to completion status. The output of synchronizing task T3a represents the logical $AND$ of the respective incoming control signals and will invoke the task T3b only when both tasks T2a and T2b generate the respective completion status signals.

**An encapsulation program for concurrent execution**

It is instructive to articulate a program implementation of a component-based program *Parabola* that can simulate a concurrent execution. To illustrate the concept, we designed a simple component *pSolver* that finds an approximate minimum of a single variable function such as parabola by repeated execution of the pSolver component,
Figure 3.4: A task encapsulation program and a trace of concurrent execution.
using the solution from the previous iteration as a new starting point. We can increase
the number of iterations by choosing a starting point far from the optimum value.
Now, we want to assign the task of finding the minima by *simultaneously* invoking
three instances of the component pSolver – and print a trace that shows a concurrent
execution. In addition, we want to (1) find the average of two minima as soon as
two components completed execution and (2) print the results. The program should
ignore the results computed by the component that required the most iterations.

We show the taskflow of the program in Figure 3.4, along with the ‘main program’
that contains the list of task to be invoked and executed, and the the actual trace
printout. Notably, the list is not unlike one would write a ‘netlist’ of hardware
components. Tasklist is defined as a directed acyclic (polar) graph, with ‘BEGIN’
and ‘END’ node at each of its poles. Each component is expressed in terms of its
predecessor components. For example, predecessor for taskD is the list of tasks { taskA, taskB, taskC }, implying that taskD is to be invoked once conditions posted
for the completion of the predecessor tasks are satisfied. Typically, one would ‘wait’
for completion of all incoming tasks. In this example, taskD is instructed to wait only
for completion of two tasks before its invocation.

Each task is scheduled for execution by its own finite-state-machine that will be
described later in the section. The trace of reported results shows that task are
clearly executed concurrently, with taskC completed first. While these are results of
simulation, we use the same example to illustrate the actual implementation of the
taskflow GUI, shown later in the thesis (see Figure 3.23).

### 3.1.3 Extending Hardware Concepts to Taskflow Programming

Compared to the programming language description of the 4-task decomposition in
Figure 3.3, the hardware description language supports a number of features not
readily supported in the programming language:

- explicit declaration of input/output arguments of the top-level module as well as
  its cells. For example, `.ix(..)` and `.ox(..)` encapsulate all input/output signals
for each cell, whereas module inputs and output are declared more explicitly.

- distinct instance names associated with invocation of each task (e.g. T2a, T2b, both implemented as replicas of cell2) – unlike the programming language description. Here, the definition of each module (cell1, cell2, ...) is part of a ‘module library’, where each module may have a number of implementations, depending the context of its use: at least one for simulation, at least one for logic optimization, at least one for placement and routing, etc.

- explicit control of task sequence execution, by way of control signals that are simply specified as as additional input/output arguments of the component itself.

- unlike the programming language, the hardware description language does not impose on the user to enter the task decomposition in any particular order. The netlist entered by the user specifies all control and data dependencies directly. In order to verify the sequence of task execution before implementing them in hardware, the user simply simulates the netlist and analyzes its behavior in terms of task execution schedule and other factors.

- ability to automatically convert a well-structured netlist-based design into a design where critical components can be independently re-tested also after their embedding in a specific design. An example of a synthesis methodology has been described in [100] and is today a routine feature in most commercial logic synthesis tools. The method basically involves the capability to multiplex local and global data to embedded components to independently test each component for manufacturing defects.

Standardized components clearly play a major role in building large and complex electronic systems. Automated layout tools can cope much better with problems increasing in their complexity when cell sizes are of uniform height, etc. In our approach, we consider an incremental approach when encapsulating each component, creating instances with standardized test data that remains with the component also
after the component has been embedded in a taskflow. We illustrate the approach with the simple examples in Figures 3.5, 3.6. Each panels in these figures illustrates a view that have of each component and its environment. The standalone view in Figure 3.5 shows a set of task instances as have been tested in the component library. Each task is invoked individually and verified.

In the local taskflow view, tasks are interconnected with task-to-task control edges for concurrent/serial execution, however data accessed by each task remains the same as the data used in the standalone view. They are merely scheduled for automated execution and testing of the control structure. A data-port multiplexor has been added to each task instances with a selector that can toggle the data access between flow/local data and the selector is clearly set to local.

In the flow taskflow view, the selector is clearly set to flow and tasks are interconnected with task-to-task control edges for concurrent/serial execution as well as data-task and task-data data edges. The input data is read from the file A1. Task data ports hold the data variables and the output data from task T5 is written to the file B4.

In the user-interaction taskflow view, task control and data dependencies are re-configured by the user. Here, user clicked on the control-edge T1⇒T2 to disable it and to set the data selector of task T2 to the value of local, overriding the global setting of flow to other tasks. The input data is now read from files A1 and A3.

The interactivity afforded to user by clicking on the control edges in the taskflow allows great many configurations to be executed. Chains of tasks can be isolated and re-tested. The approach is not unlike probing the layout of a semiconductor device under a scanning electron microscope, where voltage levels on conducting paths can be probed and verified for manufacturing defects. Moreover, by present the taskflow hierarchically, only a few details need to be reviewed at any given time.

### 3.1.4 A Taskflow Environment: Key Objectives

The introduction to the notion of blackbox/whitebox components in this section provided new insights as to their properties and relationships to corresponding hardware
In a standalone view, tasks are invoked individually by the user. Each task has at least one invocation port and at least one status port. There can be any number of input/output data ports. Data read and written by tasks in this view is local data.

In a taskflow view, tasks are invoked by other tasks except for the first task. In a taskflow, all data ports are multiplexed by way of a flow/local data selector. In this configuration, the selector is set to local, hence all tasks are tested with local data.

In this taskflow view, the data selector is set to flow globally, hence all tasks now process the flow data. The input data is read from the file A1. Task data ports hold the data variables, the output data from task T5 is written to the file B4.

Figure 3.5: Task execution views: standalone and taskflow with local or flow data.
components. For example:

- A blackbox software component is not unlike a hardware primitive: both are characterized by the distribution of I/O control and data ports and the expected behavior in terms of I/O responses.

- A hierarchy of whitebox components can be created and encapsulated to become a new whitebox or, if we so choose, a blackbox.

- Just as hardware components are scheduled for concurrent execution, so are the software components.

- When probing for structural defects in hardware, we increase the observability and controllability of components by multiplexing functional/test data on its ports. We can apply the same concepts to debugging a composition of software components.

Given these insights, we envision that the proposed taskflow GUI environment should feature the following attributes:
• A GUI that supports a hierarchical access and invocation of each component in the tree hierarchy.

• A GUI that supports, at each level of tree hierarchy, a directed dependency graph of the whitebox and blackbox components, not unlike the graphs we created manually for the simple taskflow examples in this section.

• A GUI representation that is highly interactive if the user so chooses. Most significantly, user has the option to turn on/off each task-to-task control edge, thereby be able to (1) isolate selected task in the graph and execute them independently of others, (2) provide interactive control of execution to collaborating but distributed participants.

• A GUI representation where instances of software components and their interconnections are generated automatically by extending the schema of a portable and a standardized mark-up language such as XML.

3.2 Task Primitives, Layers, and Taskflow

The abstract and layered task primitives and the taskflow as defined in this thesis, are the fundamental data structures for the proposed taskflow architecture. We introduce the definitions and graphical symbols for task node ports, edges, task nodes and data nodes in three subsections: (1) abstract task primitives used to define and create a taskflow, (2) layered task primitives used to create a hierarchical taskflow, and (3) taskgraph and datagraph used to represent task-to-task, task-to-data, data-to-task dependencies.

3.2.1 Abstract Task Primitives

Each taskflow described in this thesis can be decomposed into a total of five abstract primitive task types:

• *EndJoin/*ControlJoin (*EJ/CJ*),
- **BeginFork/ControlFork (BF/CF),**
- **DataMultiplexor (DM),**
- **FiniteStateMachineDatapath (FSMD),** and
- **BlackBoxComponent (BBC).**

While primitives EndJoin and ControlJoin are structurally equivalent, we use different names depending on the layer at which they are invoked. Similarly, BeginFork and ControlFork are structurally equivalent. Each primitive task consists of a task body and one or more ports that form a characteristic port distribution on the boundary of its body. Each task communicates with its environment by way of connectors, attached to ports. The ports hold, and can be probed, for two classes of variables: control variables or data variables. The data variables are of two types: temporary and persistent. We say that a data variable is persistent if its value is saved in a file.

Depending on the variable class, we distinguish between input/output ControlPorts and input/output/inout/outin DataPorts. When a task reads from and writes to the same port, the data port is either of type inout or outin. An inout port signifies that the task expects the data to be present before invocation and may overwrite it with new data. An outin port signifies that the data is not present before the task invocation and is generated internally by the task and the data may be reused by the task on subsequent repeated invocations.

Similarly, we distinguish between ControlConnectors and DataConnectors. Control connectors are directed edges, always connecting two control ports only. Data connectors are directed edges that may connect data to a DataPort, a DataPort to data, or a DataPort to a DataPort. Control connectors are of three types: invocation, abort, and repeat and are always considered in pairs: user-defined and program-defined. While program-defined control connectors are always in the closed state, user-defined connectors can be in either either closed state or open state, depending on the interaction by the user. Clearly, a taskflow is a complicated data structure, representing an intersection of a task graph and a data graph. We shall
<table>
<thead>
<tr>
<th><strong>TaskPrimitives</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>BlackboxComponent (BBC)</td>
</tr>
<tr>
<td>FiniteStateMachineDatapath (FSMD)</td>
</tr>
<tr>
<td>BeginFork or ControlFork (BF or CF)</td>
</tr>
<tr>
<td>EndJoin or ControlJoin (EJ or CJ)</td>
</tr>
<tr>
<td>DataMultiplexor (DM)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ControlConnectors</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>ProgramControl (port–port, always closed)</td>
</tr>
<tr>
<td>UserInvocation (port–port, opened/closed by user)</td>
</tr>
<tr>
<td>UserAbort (port–port, opened/closed by user)</td>
</tr>
<tr>
<td>UserRepeat (port–port, opened/closed by user)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>DataConnector</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>(data–port, port–data, port–port)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>ControlPorts</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>TaskInvocation (input)</td>
</tr>
<tr>
<td>TaskStatus (output)</td>
</tr>
<tr>
<td>TaskAbort (input/output)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>DataPorts</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>DataSingleton (input/output/inout/outin)</td>
</tr>
<tr>
<td>DataPair (input/output/inout/outin)</td>
</tr>
<tr>
<td>DataStack (input/output/inout/outin)</td>
</tr>
<tr>
<td>DataStackPair (input/output/inout/outin)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>DataTypes</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>temporary</td>
</tr>
<tr>
<td>persistent</td>
</tr>
</tbody>
</table>

Figure 3.7: A schema relating primitive tasks, connectors, ports, and data.

postpone the definition of task and data graphs until we raise the abstraction of the task from the primitive level to a higher level.

The informal schema in Figure 3.7 relates the primitive tasks, connectors, ports and data. A companion illustration in Figure 3.8 depicts the symbols that are used for task primitives, connectors, and data throughout this thesis. A formal schema, based on XML, that we use to implement the taskflow, is introduced in Section 3.4.
Connectors, Ports, and Data

This is a complete description of connectors, ports, and data, listed in Figures 3.7. All data port descriptions hold for both temporary and persistent data types.

ProgramControl connector is a control-edge in the taskflow control graph to which no user access is available after loading the taskflow. In a software program, such edges are created implicitly by any sequence of statements or procedure
calls. The control-edge connects the status port of a task to the invocation port of the same or another task. The control variable can have a number of states, including a state that invokes or aborts a task at the destination invocation port.

**UserInvocation connector** is a *control-edge* in the task graph whose state can be modified by the user *after* loading the taskflow. All UserInvocation connectors are created explicitly during the task decomposition and the formation of the taskflow. By default, a GUI of the taskflow displays tasks and control-edges in the enabled state. However, by clicking on the edge, user can toggle it between the enabled/disabled states. While in general, the control-edge connects the status port of a task to the invocation port of the same or another task, user also implicitly creates and enables the UserInvocation connector by directly clicking on the task to be invoked. An example where such a feature is useful is the case of a task driven by user-disabled control-edges. Such a task cannot be invoked by any task in the taskflow, but user can invoke it directly by clicking on it.

**UserAbort connector** is a *control-edge* in the task graph whose state can be modified by the user *after* loading the taskflow. Just as UserInvocation connectors, any number of UserAbort connectors may be created during the task decomposition and the formation of the taskflow. Similarly, user also implicitly creates and enables the UserAbort connector by directly clicking on the task to be aborted.

**UserRepeat connector** is a *control-edge* in the task graph whose state can be modified by the user *after* loading the taskflow. By clicking on the edge, user can disable invocation of a task that may have been scheduled for repeated invocation in a taskflow. Afterwards, user can just click on the task node manually, to repeat the invocation of the task.

**DataConnector** is a *data-edge* created in taskflow data graph *after* completing the top-down decomposition of tasks into the taskflow control graph. In a soft-
ware program, such edges are created implicitly when assigning values to variables, passing/returning argument lists between the procedure calls, etc. In the taskflow, the data connector holds the value(s) of data, associated with the respective data ports.

**TaskInvocation port** is an input control port that holds the state of the TaskInvocation variable read by the task body for appropriate action.

**TaskAbort port** is an output control port that holds the state of the TaskStatus variable written by the task body.

**TaskStatus port** is an input/output control port that connects abort edges only. An abort edge may exist between two tasks, and depending on the state of task1, task2 may be aborted by an abort signal from task2.

**DataSingleton port** is an input/output/inout/outin data port that holds the value(s) of the DataSingleton variable read/written by the task body.

**DataPair port** is a pair of input/output/inout/outin data ports that hold the value(s) of the DataPair variables read/written by the task body.

**DataStack port** is a stack of input/output/inout/outin data ports that hold the value(s) of the DataStack variables read/written by the task body.

**DataStackPair port** is a pair of input/output/inout/outin data ports that hold the value(s) of the DataStack variables read/written by the task body.

**Primitive Tasks**

This is a description of five primitive tasks listed in Figures 3.7. Notably, as shown in Figure 3.7, we associate a unique distribution of ports with each primitive task. By convention, most if not all input ports are assigned to the body boundary on the left, and most if not all output ports are assigned to the body boundary on the right. Designation for the few, if any, ports on the top or the bottom boundary of the task body is given explicitly. Data associated with all data ports can *always* be of type
temporary or persistent. All primitive tasks, except BlackBox tasks, are expected to reside on the host where the taskflow is being invoked.

**BlackBox component (BBC):** The port distribution on the boundary of this task is characterized by a single TaskInvocation control port, a single TaskStatus control port, and zero or more DataSingleton, DataStack input/output ports. This task may reside on any unix/linux/windowsNT/mac host as long as it is accessible on the Internet.

**FiniteStateMachineDatapath (FSMD):** The port distribution on the boundary of this task is characterized by a single TaskInvocation control port, a Reset control port, a single TaskStatus control port, data ports pertaining to FSMD configuration for the task, and zero or more DataSingleton, DataStack input/output ports.

**BeginFork or ControlFork (BF or CF):** The port distribution on the boundary of this task is characterized by a single TaskInvocation control port, one or more TaskStatus control ports, and zero or more DataSingleton and DataStack input/output ports. The purpose of the BeginFork or ControlFork task is to analyze incoming control variable and data and to create appropriate control variables that decide the invocation of all tasks driven by the respective TaskStatus ports.

**EndJoin or ControlJoin (EJ or CJ):** The port distribution on the boundary of this task is characterized by one or more TaskInvocation control ports, a single TaskStatus control ports, zero or more Abort control ports, and zero or more DataSingleton, DataStack input/output ports. The purpose of the EndJoin or ControlJoin task is to analyze incoming control variable(s) and data and to create appropriate control variable that decides the invocation of the next task driven by the respective TaskStatus ports.

**DataMultiplexor(DM):** The port distribution on the boundary of this task is characterized by one or more TaskInvocation control ports and zero or more DataSingletonPair and zero or more DataStackPair ports. As to output ports, there
are zero or more DataSingleton and zero or more DataStack ports. The purpose of the DataMultiplexor task is to select data local or flow data before executing the task.

3.2.2 Layered Task Primitives

The interconnection of the five task primitives defined in section 3.2.1 is subject to few simple and well-defined rules. Each of the interconnection rules defines a taskflow layer. We introduce a total of five types of layers: an encapsulation layer for BlackBox and WhiteBox, a task instance layer for BlackBox and WhiteBox, and a component layer for the WhiteBox with the ports on the boundary of the WhiteBox having the same generic distribution we associate with the BlackBox. In addition to formal description of layers in the text below, the process of developing these layers is also illustrated in Figure 3.9.

**Encapsulation layer (BlackBox):** This layer encapsulates the FSMD and the BlackBox abstract primitive tasks. On the layer boundary, we connect the Reset, Invoke/Abort, and Completion Status ports to corresponding FSMD ports. Data ports on the layer boundary are connected to corresponding data ports on the FSMD and the BlackBox. Next, we interconnect the FSMD and the BlackBox using two control connectors, one carrying an invocation signal \( t_{\text{act}} \) from the FSMD to the BlackBox, the other carrying the task completion signal \( t_{\text{con}} \) from the BlackBox to the FSMD. The use of such handshaking protocol is the characteristic feature of the FSMD itself [88, 85]. In this case, the BlackBox clearly represents an extension of the Datapath.

**Task instance layer (BlackBox):** This layer instantiates the encapsulated BlackBox, and the abstract task primitives ControlJoin (CJ), ControlFork (CF), and DataMultiplexor (DM). On the layer boundary, abutted by CJ and DM on the left, we find the control ports that connect invocation and abort edges from incoming tasks, the Reset port, and the flowData/localData control port that controls DM to select data generated during the execution of the flow, or the
An encapsulated black/whitebox component (BBC/WBC) represents an arrangement with FSMD (finite-state-machine with datapath). The black/whitebox component is simply an extension of data path, communicating with FSM by way of two handshaking signals, \( t_{\text{inv}} \) and \( t_{\text{com}} \). Control and data port symbols defined in Figure 3.8 complete the definition introduced in Figure 3.1.

A whitebox component is a polar graph of task instances and data, with the BeginFork and EndJoin primitive tasks representing the source node and the sink node respectively. Unlike whitebox definition introduced in Figure 3.1, there are no decision nodes. These nodes have been split and are now within each task instance, defined below.

A task (or taskflow) instance is an arrangement of an encapsulated blackbox or whitebox component and task primitives: ControlJoin, ControlFork, and Data-Multiplexor – with data attached to its ports. This definition extends the one introduced in Figure 3.1.

Figure 3.9: Layered primitives of the task model.
pre-defined local data, associated with the stand-alone task instance. The input data ports on the layer boundary are associated with DM, the output data ports on the layer boundary right side represent the extensions of the output data ports of the encapsulated BlackBox. On the layer boundary, abutted by CF on the right, we also find control ports that connect invocation and abort edges to outgoing tasks.

Within the task instance layer, we connect the control status port of ControlJoin, the encapsulated BlackBox, and the ControlFork using two control connectors, one from the status port of ControlJoin to control invocation port of the encapsulated BlackBox, and the other from the status port of the the encapsulated BlackBox to the invocation port of the ControlFork. Finally, we connect the interior input/output data ports of the abstract task primitives in the manner required for specific task instance.

**Component layer (WhiteBox):** This layer represents the interconnection of task instances, data nodes, and two special abstract task primitives: BeginFork and EndJoin. As we said earlier, BeginFork and ControlFork are equivalent, and so are EndJoin and ControlJoin. However, for clarity we use the name BeginFork rather than ControlFork, and EndJoin rather than ControlJoin when describing the graph formed within the WhiteBox component layer.

The WhiteBox component layer defines the taskflow as an intersection of two graphs: a TaskGraph and a DataGraph where the TaskGraph is a polar DAG (directed acyclic graph), with two distinguished vertices, source and a sink which we name BeginFork and EndJoin in this thesis. In polar DAG, all vertices are reachable from the source, and the sink is reachable from all vertices. Hence, the source represents the first task (BeginFork), and the sink represents the last task (EndJoin). As we elaborate in the next section, a TaskGraph is a DAG only when viewed in terms of edges of the same type.

The ports on the boundary of the WhiteBox have the same generic distribution we associate with the BlackBox: a single TaskInvocation control port,
a single TaskStatus control port, and zero or more DataSingleton, DataStack input/output ports. Unlike the BlackBox that represents a single component residing on any unix/linux/windowsNT/mac host, the WhiteBox is a composition of BlackBox and WhiteBox components. The encapsulation and the instantiation of WhiteBox components follows the same rules already described for the BlackBox above and summarized only briefly next.

**Encapsulation layer (WhiteBox):** This layer encapsulates the FSMD and the WhiteBox component as defined above. The interconnection rules are the same as those defined for the BlackBox.

**Task instance layer (WhiteBox):** This layer instantiates the encapsulated WhiteBox, and the abstract task primitives ControlJoin (CJ), ControlFork (CF), and DataMultiplexor (DM). The interconnection rules are the same as those defined for the BlackBox.

An illustrative summary of these definitions is shown in Figure 3.9.

### 3.2.3 Taskflow as a TaskGraph and a DataGraph

In any task-driven project, there is a hierarchy of tasks and task/data dependencies. In this section, we formalize the representation of a taskflow at the level of (hierarchical) task instances created by the user in the process of task definition and composition. In the sections that follow, we use this representation to (1) formalize the taskflow scheduling, and (2) rendering any taskflow as a hierarchy of task trees and task graphs.

A taskflow represents a WhiteBox component within the hierarchy of taskflow decompositions. We model it as a directed graph with tasks and data as vertices, and directed task-to-task control-edges and data-to-task or task-to-data data-edges. Each task is represented as a BlackBox task instance or a WhiteBox task instance. The control edges are of three types: invocation, abort, and repeat, and are always considered in pairs: user-defined and program-defined. While program-defined control
edges are always in the closed state, user-defined edges can be in either either closed state or open state, depending on the interaction by the user.

Any taskflow is actually an intersection of two graphs: a TaskGraph and a DataGraph. The construction of DataGraph is simply in terms of data-to-task or task-to-data data-edges. However, there are three views of the TaskGraph, depending on the edge type. When viewed in terms of invocation edges, the TaskGraph is a polar DAG (directed acyclic graph), with two distinguished vertices, source and a sink, named BeginFork and EndJoin. In a polar DAG, all vertices are reachable from the source, and the sink is reachable from all vertices. Hence, the source represents the first task (BeginFork), and the sink represents the last task (EndJoin).

When viewed in terms of task-to-task abort edges, a cycle is never formed. Only when viewed in terms of the repeat edge, implicitly associated with each task, a self-loop is being formed for each task.

We complement the description of the taskflow model with two illustrations in Figure 3.10. The top-most illustration depicts a view of task instance, with attributes that guide the actual GUI implementation, such as buttons to reset/invoke/abort/skip the task, and the types of user-defined control edges. The illustration below represents a re-arrangement of the WhiteBox component, introduced in Figure 3.9. Here, the BeginFork task and EndJoin task abutt the respective component layer, a prelude to the GUI arrangement later. The intersection of a TaskGraph and a DataGraph can be readily recognized in this taskflow example.

### 3.3 Taskflow Scheduling

As defined in the preceding section, a WhiteBox component is an intersection of a TaskGraph and a DataGraph. We have to encapsulate the WhiteBox component before we can execute it – the invocation signal for the WhiteBox component is provided by the FSM after WhiteBox encapsulation. Taskflow scheduling thus involves the execution of the WhiteBox component, i.e. the scheduling of all task instances contained in this component.

Since the instances of encapsulated components in the taskflow execute asyn-
Each whitebox component, accessible via GUI, contains two or more task instances. Each task instance can be accessed by the user directly: to be reset, invoked, repeated, aborted (if executing), and skipped. Alternatively, each task instance can be invoked/aborted by other tasks.

A whitebox component is a polar graph of task instances and data, with the BeginFork and EndJoin primitive tasks representing the source node and the sink node respectively. We model the taskflow it represents as an intersection of a task graph and a data graph. The task graph, with user-controllable invocation/abort/repeat edges as shown, is rendered by the GUI.

Figure 3.10: A taskflow model: an intersection of a task graph and a data graph.

chronously, the FSM we use to schedule each component must also be asynchronous. To ensure the deterministic operation of an asynchronous FSM, we implement it as a pulse-mode FSM and give a brief overview of such a machine. Next, we introduce the encapsulated component architecture model as an extension of the FSMD model and conclude with a taskflow scheduling algorithm.
3.3.1 Background: Pulse-Mode FSM

Most scheduling algorithms in automated hardware synthesis are based on the assumption of a synchronous finite-state-machine (FSM), i.e. one whose state transitions are controlled by a clock. To ensure the deterministic operation of an asynchronous FSM, we must prohibit simultaneous changes on two or more of its inputs [101]. By making the inputs into pulses of short duration and maintaining that pulses occur on the FSM inputs only one at a time, the design and operation of a pulse-mode FSM parallels that of the synchronous machine, with few adjustments. Notably, only the uncomplemented form of the input signal may be used.

To illustrate key principles, we first consider a synchronous 2-state FSM used by the adaptive scheduling algorithm as reported in [84, 102]. Here, we schedule a *synchronous* BlackBox component, shown on the left-half of Figure 3.11. If the BlackBox component were controlled by a synchronizing clock, the simple 2-state FSM could be used to encapsulate it. On the other hand, if the BlackBox component is asynchronous, the pulse-mode version of the FSM should be used. The BlackBox outputs a narrow pulse to the gated inputs of the flip-flop to induce a transition upon completed execution of the component.

In practical terms, one would want to encapsulate a component with a FSM that

![Simple 2-state FSMD models of encapsulating and scheduling a task.](image)

On the left, we have of a 2-state Mealy synchronous FSM interacting with a BBC. Both are synchronized with the same clock. On the right, we have a 2-state Mealy *pulse mode* FSM interacting with a BBC. Here, both FSM and BBC execute asynchronously.
has more than two states. We show an example of a 4-state implementation in Figure 3.12. The states are: Waiting, Executing, Completed, and Done. The reset pulse (R) always returns the FSM into the waiting state. If the invocation pulse $P_i$ arrives, we have a transition $W \Rightarrow X$ and upon completion of execution (pulse $t_e$, generated when $t_{con} = 1$), the FSM transitions from $X \Rightarrow C$. If the ‘repeat’ pulse ($t_r$) arrives, FSM transitions from $C \Rightarrow D$.

In general, BBC is invoked when the Boolean expression $t_{inv} \equiv Q_X = P_i.W + t_r.C$ is satisfied. Clearly, this arrangement is more useful; now we can also schedule a data-dependent iteration of the component invocation. The presence of a counter and comparator makes it very clear that the black box is merely an extension of a Datapath. The software implementation of the FSMD in Figure 3.12 is based on the pull model of connection-oriented programming [73], which follows the conventional approach of ‘pulling’ the information in as needed, appears shorter, and is easier to read. However, in our implementation of the FSMD, we follow the more advanced push model of connection-oriented programming where the information is ‘pushed’ out as it arises resulting in propagation of event or notification. The reported performance of the scheduler in Chapter 5 confirms the efficiency of the current implementation under a wide range of conditions.

### 3.3.2 Task Instance Architecture

The architecture of the task instance is based on the arrangement of the abstract task primitives FSMD, ControlJoin (CJ), DataMultiplexor (DM), ControlFork (CF) and BlackBox/WhiteBox components introduced in Figure 3.9. These primitives and components form a network as shown in Figure 3.13.

The task primitives ControlJoin, DataMultiplexor, and ControlFork represent combinational logic and can be described in terms of Boolean equations. In Figure 3.13, we present the equations in a tabular form as formulas. While these are complete for the DataMultiplexor and the ControlFork, we list only the default conditions for ControlJoin. The purpose of the ControlJoin is to synchronize the status of predecessor tasks before invoking the current task instance. A number of such
We control the execution of a black-box component with a 4-state pulse-mode FSM. The shown state-transition table is implemented in hardware and software. Here, BBC is invoked when the Boolean expression $t_{\text{inv}} \equiv Q_X = P_t.W + t_r.C$ is satisfied.

Figure 3.12: A pulse-mode FSMD model of hardware and software task encapsulation.
The architecture of the task instance is based on the arrangement of the abstract task primitives FSMD, ControlJoin (CJ), DataMultiplexer (DM), ControlFork (CF) and BlackBox/WhiteBox components introduced in Fig. 3.9. The functional descriptions of CJ, DM and CF are given below. The description of FSMD is summarized in Fig. 3.15.

(a) ControlJoin conditions:

Default join:

\[
\begin{align*}
P_I &= \prod_{m=1}^{M} E_{f,m} \cdot Q_{V_m} \\
P_S &= \sum_{m=1}^{M} E_{f,m} \cdot (Q_{N_m} + Q_{S_m} + Q_{T_m}) \\
P_A &= \sum_{m=1}^{P} E_{a,m} \cdot Q_{V_m}
\end{align*}
\]

For other representative join conditions, see Figure 3.14

(b) Data multiplexer:

\[
\begin{array}{c|c|c}
E_g & E_{I_e} & D_{I_f} \\
\hline
0 & x & D_{I_1} \\
1 & 0 & D_{I_{1_0}} \\
1 & 1 & D_{I_{1_1}}
\end{array}
\]

(c) ControlFork table:

\[
\begin{array}{cccccccc|cccccccc}
\text{Inputs} & Q_E & Q_X & Q_C & Q_T & Q_A & Q_S & Q_D & D_{O_n} & Q_{E_n} & Q_{X_n} & Q_{C_n} & Q_{T_n} & Q_{A_n} & Q_{S_n} & Q_{V_n} & Q_{N_n} \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & x & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & x & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & x & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & x & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{array}
\]

† user-specified fork condition such as, ‘size(D_{O_n}) > 128’, is true
‡ user-specified fork condition such as, ‘size(D_{O_n}) > 128’, is false

Figure 3.13: The architecture of the task instance.
conditions may exist, depending on the purpose of the current task; a representative set of alternative ControlJoin conditions is listed in Figure 3.14. These conditions are further analyzed in terms of the reported experiments in Section 3.6.

The abstract task primitive FSMD is at the very core of the proposed task instance architecture. The example of the simple 4-state FSMD introduced in the preceding section served to illustrate its role and the basic concepts. In the current FSMD implementation, we extend the number of its states to eight. The FSMD is described in terms of the state-transition table and a Datapath table in Figure 3.15. In particular, the invocation signal for BBC (or WBC) is generated by evaluating $t_{\text{inv}} = Q_X = t_b.E$. The BBC (or WBC) completion signal $t_{\text{com}}$ is returned as the output of the Datapath. We expand on tables and equations in Figures 3.13, 3.14, and 3.15 in more detail next.

**ControlJoin Conditions (Default)**

The ControlJoin primitive generates one of the three types of program signal pulses: (1) an invocation pulse $P_I$, (2) a skip pulse $P_S$, and (3) an abort pulse $P_A$. The first two pulses are generated based on the various states of its M predecessor tasks as well as the user-configurations of the control-edges represented by $E_{f_m}$. Similarly, the abort pulse is based on the various states of its P predecessor tasks as well as the user-configurations of the abort-edges represented by $E_{a_m}$. As shown in Figure 3.13 (a), $P_I$ pulse is generated when all of its predecessor tasks have a ‘valid’ state $Q_{V_m}$ and when all the control-edges $E_{f_m}$ are enabled. On the other hand, $P_S$ pulse is generated when any of its predecessor task is in either ‘not-valid’, ‘skipped’ or ‘timed-out’ state and when the corresponding control-edge $E_{f_m}$ is enabled. A $P_A$ pulse is generated when any one of its predecessor task is in ‘valid’ state and when the corresponding abort-edge $E_{a_m}$ is enabled. These three equations specify the default join condition for the CJ primitive.
Representative ControlJoin Conditions

In addition to the above default join condition, it is possible to specify more complex join conditions. Figure 3.14 shows the equations for few such conditions. Consider the synchronize merge condition. The invocation pulse and the skip pulse is slightly modified from those for the default join condition whereas the abort pulse is the same as the one for default join. Here, the $P_I$ pulse is basically generated whenever all of its predecessor tasks are either in ‘valid’ state or ‘skipped’ state and when all the corresponding control-edges $E_{f_m}$ are enabled. Similarly, the $P_S$ pulse is generated when any of its predecessor task is in either ‘not-valid’, or ‘timed-out’ state and when the corresponding control-edge $E_{f_m}$ is enabled.

Additional conditions listed in Figure 3.14 are further analyzed in terms of the reported experiments in Section 3.6.

Data Multiplexor

This primitive is used to switch between local data and flow data during taskflow execution. When the user-configured global signal $E_g$ is disabled, it selects the local data as represented by $D_{fi}$ in Figure 3.13 (b). When $E_g$ is enabled, it is essentially in flow data mode, however, users can still selectively switch to local data coming from certain predecessor tasks by disabling the corresponding control-edge $E_{fi}$. Thus, the DataMux primitive selects the flow data only when $E_g$ and $E_{fi}$ are both enabled.

ControlFork Table

The ControlFork primitive is a combinational logic used to output the state of the FSMD and when the task completes, it also validates user-specified condition, if any, for the output data $D_{O_n}$ and generates a corresponding ‘valid’ or ‘not-valid’ output state. For example, in Figure 3.13 (c), the user specified condition is ‘$size(D_{O_n}) > 128$’. Accordingly, the last two rows in the table shows that it generates a ‘valid’ state $Q_{V_n}$ when the condition evaluates to true and it generates a ‘not-valid’ state $Q_{N_n}$ when the condition evaluates to false.
For default join condition, see Figure 3.13 (b)

Synchronize merge:

\[
\begin{align*}
P_I & = \prod_{m=1}^{M} E_{f_m} \cdot (Q_{V_m} + Q_{S_m}) \\
P_S & = \sum_{m=1}^{M} E_{f_m} \cdot (Q_{N_m} + Q_{T_m}) \\
P_A & = \sum_{m=1}^{\mu} E_{a_m} \cdot Q_{V_m}
\end{align*}
\]

Milestone:

\[
\begin{align*}
P_I & = \prod_{m=1}^{M-X} E_{f_m} \cdot Q_{V_m} \cdot \prod_{m=X+1}^{M} E_{f_m} \cdot (Q_{E_m} + Q_{X_m}) \\
P_S & = \sum_{m=1}^{M-X} E_{f_m} \cdot (Q_{N_m} + Q_{S_m} + Q_{T_m}) + \\
& \sum_{m=X+1}^{M} E_{f_m} \cdot (Q_{N_m} + Q_{V_m} + Q_{T_m}) \\
P_A & = \sum_{m=1}^{\mu} E_{a_m} \cdot Q_{V_m}
\end{align*}
\]

Deferred choice:

\[
\begin{align*}
P_I & = \prod_{m=1}^{M} E_{f_m} \cdot Q_{V_m} \\
P_S & = \sum_{m=1}^{M} E_{f_m} \cdot (Q_{N_m} + Q_{S_m} + Q_{T_m}) \\
P_A & = \sum_{m=1}^{\mu} E_{a_m} \cdot (Q_{E_m} + Q_{X_m})
\end{align*}
\]

L-out-of-M join (shown for 2-out-of-3):

\[
\begin{align*}
P_I & = (E_{f_1} \cdot Q_{V_1}) \cdot (E_{f_2} \cdot Q_{V_2}) + (E_{f_1} \cdot Q_{V_1}) \cdot (E_{f_3} \cdot Q_{V_3}) + (E_{f_2} \cdot Q_{V_2}) \cdot (E_{f_3} \cdot Q_{V_3}) \\
P_S & = (E_{f_1} \cdot (Q_{N_1} + Q_{S_1}) \cdot E_{f_2} \cdot (Q_{N_2} + Q_{S_2})) + (E_{f_1} \cdot (Q_{N_1} + Q_{S_1}) \cdot E_{f_3} \cdot (Q_{N_3} + Q_{S_3})) + (E_{f_2} \cdot (Q_{N_2} + Q_{S_2}) \cdot E_{f_3} \cdot (Q_{N_3} + Q_{S_3})) \\
P_A & = \sum_{m=1}^{\mu} E_{a_m} \cdot Q_{V_m}
\end{align*}
\]

L-out-of-M join (shown for 2-out-of-3) with abort:

\[
\begin{align*}
P_I & = (E_{f_1} \cdot Q_{V_1}) \cdot (E_{f_2} \cdot Q_{V_2}) + (E_{f_1} \cdot Q_{V_1}) \cdot (E_{f_3} \cdot Q_{V_3}) + (E_{f_2} \cdot Q_{V_2}) \cdot (E_{f_3} \cdot Q_{V_3}) \\
P_S & = (E_{f_1} \cdot (Q_{N_1} + Q_{S_1}) \cdot E_{f_2} \cdot (Q_{N_2} + Q_{S_2})) + (E_{f_1} \cdot (Q_{N_1} + Q_{S_1}) \cdot E_{f_3} \cdot (Q_{N_3} + Q_{S_3})) + (E_{f_2} \cdot (Q_{N_2} + Q_{S_2}) \cdot E_{f_3} \cdot (Q_{N_3} + Q_{S_3})) \\
P_A & = \sum_{m=1}^{\mu} E_{a_m} \cdot Q_{V_m}
\end{align*}
\]

Foreach \( m \) in \( M \) incoming tasks:

\[
P_{A_m} = E_{a_m} \cdot (Q_{E_k} + Q_{X_k})
\]

Figure 3.14: Summary of representative ControlJoin conditions.

State Transition Table

The FSM has eight states corresponding to different stages of execution and is always initialized to ‘waiting’ state W. The FSM changes to a new state depending on its
The Finite State Machine with Data-path (FSMD) that implements the architecture of the task instance in Figure 3.13 is described in terms of a state-transition table and a datapath table below.

In particular, the invocation signal for BBC (or WBC) is generated by evaluating $t_{inv} \equiv Q_X = t_h.E$. The BBC (or WBC) completion signal $t_{com}$ is returned as the output of the datapath. For more details, see text.

(a) State transition table

<table>
<thead>
<tr>
<th>Inputs/NS</th>
<th>Outputs</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_S$</td>
<td>$R$</td>
</tr>
<tr>
<td>-----------</td>
<td>---------</td>
</tr>
<tr>
<td>W</td>
<td>W</td>
</tr>
<tr>
<td>E</td>
<td>W</td>
</tr>
<tr>
<td>X</td>
<td>W</td>
</tr>
<tr>
<td>C</td>
<td>W</td>
</tr>
<tr>
<td>D</td>
<td>W</td>
</tr>
<tr>
<td>A</td>
<td>W</td>
</tr>
<tr>
<td>T</td>
<td>W</td>
</tr>
<tr>
<td>S</td>
<td>W</td>
</tr>
</tbody>
</table>

(b) Datapath table

<table>
<thead>
<tr>
<th>Outputs (pulses)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$D_{new}$</td>
</tr>
<tr>
<td>------------------</td>
</tr>
<tr>
<td>x x x x x x</td>
</tr>
<tr>
<td>0 x x x x x x</td>
</tr>
<tr>
<td>1 x x x x x x</td>
</tr>
<tr>
<td>1 x x x x x x</td>
</tr>
<tr>
<td>1 1 x 1 x x</td>
</tr>
<tr>
<td>1 1 x 1 x x</td>
</tr>
<tr>
<td>1 1 0 1 1 x</td>
</tr>
<tr>
<td>1 1 0 1 1 x</td>
</tr>
<tr>
<td>x x x x x x</td>
</tr>
<tr>
<td>x x x x x x</td>
</tr>
<tr>
<td>x x x x x x</td>
</tr>
<tr>
<td>x x x x x x</td>
</tr>
<tr>
<td>x x x x x x</td>
</tr>
<tr>
<td>x x x x x x</td>
</tr>
<tr>
<td>x x x x x x</td>
</tr>
<tr>
<td>x x x x x x</td>
</tr>
<tr>
<td>all remaining entries</td>
</tr>
</tbody>
</table>

† user-specified repeat condition such as, $\text{cost}(D_{out}) > 10$, is true
‡ user-specified repeat condition such as, $\text{cost}(D_{out}) > 10$, is false

Figure 3.15: FSMD specification for the architecture of an encapsulated task.
present state and the type of input signal it receives. For example, when the FSMD is in ‘enabled’ state $E$: (1) it changes to ‘executing’ state $X$ when the host is available for processing tasks, indicated by the $t_h$ pulse, (2) it changes to ‘done’ state $D$ when the input data is older than it output data, indicated by the $t_d$ pulse, (3) it changes to ‘skipped’ state $S$ when the executable program is not available on the host, indicated by the $t_s$ pulse, and (4) it changes to ‘abort’ state $A$ when it receives the $P_a$ pulse from CJ. Figure 3.15 (a) shows the entire state transition table for the FSM. The output signals $Q_E$ through $Q_S$ are generated directly based on the current state of FSM.

Datapath Table

The Datapath module of FSMD generates various signal pulses based on the current state of FSM and several other inputs. As shown in Figure 3.15 (b), it consists of (1) five inputs associated with BBC component: $O_v$ represents that owner of BBC is verified, $H_s$ represents that host of BBC exists, $H_b$ represents that host of BBC is busy, $P_c$ represents that program of BBC exists, and $D_{new}$, represents that input data of BBC is new, (2) three state inputs from FSM: $Q_E$, $Q_X$, and $Q_C$, (3) three inputs for user-configured states: $E_s$ represents always skip execution, $E_c$ represents always check for input/output data, and $E_r$ represents enabled repeat-edge, and (4) one input for user-specified data-dependent repeat condition $D_{all}$. Based on these inputs, the Datapath module generates six output signal pulses, which are used by the FSM to change to a new state: (1) $t_{com}$ represents task completion signal, (2) $t_d$ represents task done signal, (3) $t_s$ represents task skipped signal, (4) $t_n$ represents task need-not-re-execute signal, (5) $t_h$ represents task host available signal, and (6) $t_r$ represents task repeat signal. Figure 3.15 (b) shows the entire Datapath table and the various conditions under which the output signal pulses are generated.

3.3.3 Taskflow Scheduling Algorithm

The invocation of any task instance $T_k$ is subject to evaluation of a number of control signals as well as data values. Consider the task instance architecture in Figure 3.13.
Major evaluations take place within the ControlJoin, FSMD and ControlFork.

Let \( m \in \text{pred}(T_k) \) designate \( m \) control-edge predecessors of task \( T_k \); \( p \in \text{abort}(T_k) \) designate \( p \) abort-edge incoming tasks of task \( T_k \) (these could include successors of \( T_k \)); and let \( \text{CJ}_k(Q_m, \mathcal{E}_{f_m}, \mathcal{E}_{a_p}) \) designate the assignments evaluated by the ControlJoin, where \( Q_m = \{Q_{E_m}, Q_{X_m}, Q_{C_m}, Q_{T_m}, Q_{A_m}, Q_{S_m}, Q_{V_m}, Q_{N_m}\} \) represents the states of the predecessor task \( Q_m \), \( \mathcal{E}_{f_m} \) represents the state of the forward (invocation) user-configured control-edge; and \( \mathcal{E}_{a_p} \) represents the state of the user-configured abort-edge. Then

\[
\{P_k, P_{A_k}, P_{S_k}\} \equiv \text{CJ}_k(Q_m, \mathcal{E}_{f_m}, \mathcal{E}_{a_p}) \quad \forall m \in \text{pred}(T_k), \forall p \in \text{abort}(T_k)
\]

where \( \{P_k, P_{A_k}, P_{S_k}\} \) represent the invocation, abort, or skip signal as the input to FSMD\(_k\).

As the state of the task \( T_k \) is changing, it is being evaluated by the ControlFork as the assignment

\[
Q_n \equiv \text{CF}_k(Q_k, D_{O_k}) \quad \forall n \in \text{succ}(T_k)
\]

where \( n \in \text{succ}(T_k) \) designates \( n \) control-edge successors of task \( T_k \);

\[
Q_n = \{Q_{E_n}, Q_{X_n}, Q_{C_n}, Q_{T_n}, Q_{A_n}, Q_{S_n}, Q_{V_n}, Q_{N_n}\};
\]

\[
Q_k = \{Q_{W_k}, Q_{E_k}, Q_{X_k}, Q_{C_k}, Q_{T_k}, Q_{A_k}, Q_{S_k}, Q_{D_k}\}; \quad \text{and}
\]

\( D_{O_k} \) is the set of output data values produced by task \( T_k \).

The state of the task \( T_k \) is changing according to the inputs and the state transition table in Figure 3.15. The overall scheduling algorithm that takes the transition table and the Datapath table of the FSMD\(_k\) into account is outlined in Figure 3.16. Nominally, the user will initiate the invocation of the BeginFork task, i.e. task for which \( k = 0 \), \( T_0 \). Tasks that can be executed concurrently – if the host is available (i.e. \( t_{h_k} = 1 \)) – for each scheduled task. A new task \( k \) will be invoked only if the synchronizing conditions, evaluated by a task-specific \( \text{CJ}_k \) will have been satisfied. The last task to be evaluated will always be the EndJoin task, since the TaskGraph is polar.
For each task $T_k$ ($k > 0$) {  
  /* evaluate ControlJoin conditions */  
  \{P_l, P_a, P_s\} \equiv CJ_k(Q_m, \mathcal{E}_f, \mathcal{E}_a) \quad \forall m \in \text{pred}(T_k), \forall p \in \text{abort}(T_k)  
  \text{where}  
  \begin{align*}
  Q_m &= \{Q_{E_m}, Q_{X_m}, Q_{C_m}, Q_{T_m}, Q_{A_m}, Q_{S_m}, Q_{V_m}, Q_{N_m}\}  
  \end{align*}  
  
  /* evaluate FSMD$_k$ conditions */  
  if $P_l = 1$ and $W_k = 1$ then  
  transition to state $E_k$  
  if $t_{h_k} = 1$ then  
  transition to state $X_k$ and generate invocation signal  
  for BBC$_k$/WBC$_k$: $t_{inv} = Q_{X_k} = t_{h_k}.E_k$  
  if $t_{com} = 1$ (asserted on completion of BBC$_k$/WBC$_k$)  
  transition to state $C_k$  
  if $t_{r_k} = 1$ (repeat signal, asserted by datapath)  
  transition to state $E_k$ where another invocation  
  signal $t_{inv}$ may be generated, given that $t_{h_k} = 1$, etc.  
  if $t_{d_k} = 1$ (done signal, asserted by datapath)  
  transition to state $D_k$  
  which is the nominal exit state for FSMD$_k$ input $P_{l_k} = 1$.  
  if ($P_a = 1$ and ($E_k = 1$ or $X_k = 1$)) then  
  transition to state $A_k$  
  which is the nominal exit state for FSMD$_k$ input $P_{a_k} = 1$.  
  if ($P_s = 1$ and $W_k = 1$) or ($t_{s_k} = 1$ and $E_k = 1$) then  
  transition to state $S_k$  
  which is the nominal exit state for FSMD$_k$ input $P_{s_k} = 1$ or  
  input $t_{s_k} = 1$ (generated by datapath)  
  /* evaluate ControlFork conditions */  
  $Q_n \equiv CF_k(Q_k, D_{O_k}) \quad \forall n \in \text{succ}(T_k)$  
  \text{where}  
  \begin{align*}
  D_{O_k} &= \text{the set of output data values produced by task } T_k, \text{ and}  
  Q_n &= \{Q_{E_n}, Q_{X_n}, Q_{C_n}, Q_{T_n}, Q_{A_n}, Q_{S_n}, Q_{V_n}, Q_{N_n}\}  
  Q_k &= \{Q_{W_k}, Q_{E_k}, Q_{X_k}, Q_{C_k}, Q_{T_k}, Q_{A_k}, Q_{S_k}, Q_{D_k}\}  
  \end{align*}  
  
}  

Figure 3.16: Outline of the scheduling algorithm for task $T_k$ and its successors.
3.4 XML for Taskflow Environment

In this section we look at the implementation interfaces that can be used by the user to construct taskflows. We first describe the schema used for the taskflow environment and then discuss its representation in XML.

3.4.1 Taskflow Schema

The schema we use to construct a taskflow, consists of mainly two layers, an encapsulated single or multi task layer and a task instance layer in Section 3.3. The schema for task instance layer is identical to the conceptual description. However, the schema for encapsulated task layer is composed of two separate components: (1) a single or multi task definition layer, and (2) a single or multi task body layer. Such a distinction helps in separating the task API from its body declaration, which can be very detailed. The definition layer can thus be considered as an API for the task; it is this layer that should be readily accessible.

Figure 3.17 shows the structure of the taskflow schema. The single-task and multi-task definition layers are identical to the extent that both contain a list of input, inout, outin and output ports. The type of a specific port depends on whether the data associated with a task is an input or an output. Many times, a task may read and write to the same data location, in which case the data port is either of type inout or outin. An inout port signifies that the task expects the data to be present before invocation and may overwrite it with new data. On the other hand, an outin port signifies that the data is not present before the task invocation and is generated internally by the task and the data may be reused by the task on subsequent repeated invocations.

In addition to the common port list types, the multi-task definition layer contains two more elements: (1) a TaskList that enumerates the number of encapsulated tasks occurring in the flow, and (2) a TaskGraph that specifies the directed task-to-task connectivity with control-edges. These definitions then form the API for the task layer.
Figure 3.17: A schema for XML representation of taskflow layers.
The task body layer, as the name suggests, contains more detailed information about the specific layer. The `SingleTaskBody` contains exactly three elements: `BeginFork`, `BlackBoxComponent` and `EndJoin`. Additionally, the `BeginFork` task is configured by the user to determine as to under what conditions should the blackbox component task be bypassed. Typical conditions include functionalities to invoke tasks based on time-stamps of input/output data, presence or absence of certain data files, etc. As for the blackbox component task, the user needs to specify the details necessary to invoke the external program, namely: (1) the program name, (2) the command-line arguments, (3) the host and directory location of where to invoke the program, and (4) the protocol service used to access the program, such as telnet, ftp, ssh, http, etc.

The description of `MultiTaskBody` has three primitive task elements `BeginFork`, `DataGraph` and `EndJoin` and one or more `TaskInstance` layers. The function of the `BeginFork` and `EndJoin` elements is same as for single task body layer, whereas the `DataGraph` element is used to specify the task-to-data and data-to-task data edges for the taskflow. The task instance layer is described next.

Once the encapsulated tasks have been created for either a multi-task or a single task, a `TaskInstance` layer can be created by adding a `ControlJoin`, `DataMux` and a `ControlFork` elements to it. An instance of a task is specific to the data used in invocation of the encapsulated task and several such instances can be easily created for each set of data.

Finally, it is necessary to have a `MainTask` layer which allows us to select and invoke the specific task instances from a large library. `TaskMain` element is similar to an encapsulated multi-task, it consists of the following elements: (1) a `TaskInstanceList` element that represents a list of task instances which need to be invoked at the , and (2) a `TaskGraph` that specifies the directed task-to-task connectivity with control-edges, and (3) a `BeginFork`, an `EndJoin` and one or more number of task instances created in the main layer for invocation. The main difference between an encapsulated multi-task and the main task is that the latter does not have any flow data dependencies. So each task instance is basically independent of all other task instances and the task graph merely determines the sequence of execution.
3.4.2 XML Representation

We now discuss the XML format that we use to construct taskflows. Rather than choosing our own syntax and format for describing taskflow in the taskflow environment, we decided to leverage the current XML technology [97, 103, 104, 105, 106, 107] to effectively describe our taskflows. The main advantages of using XML technology is the wide availability of various xml tools, such as xml-parsers, xml-validators and xml-editors on the Internet [108]. This allows the user to construct and edit the taskflow environment with their tools of choice.

The current limitation of using the XML technology is that it is not fully mature yet and the XML specifications are continuously evolving. For example, the specification of XML schemas [104, 109] is not yet finalized and is still a candidate recommendation only. We initially started out by using DTDs (document type definitions) [104], but soon realized the limitations of DTDs, which include: (1) DTDs are not easily extensible, (2) only one DTD may be associated with each document, (3) DTDs do not work well with namespaces, (4) non-XML syntax, (5) weak data typing, etc. Currently, we are using the more flexible XML schemas for the taskflow environment.

Figure 3.18 shows the XML schema specification for the multi-task definition layer. It is declared as an element of complex type with name MultiTaskDefn. It contains a sequence of seven elements within it, including Title, Description, InputList, InOutList, OutInList, OutputList, TaskList and TaskGraph. However, each of this element is optional and is specified using the ‘minOccurs’ and ‘maxOccurs’ attributes for the ‘xsdsquence’ tag. The first six elements are of type defined elsewhere in the schema specification, whereas the seventh element TaskGraph is of type textOnly and is used to describe the control-edges for the taskflow. In addition to these elements, the MTD layer has a ‘name’ attribute which consists of a string of simpleType. This name is used to refer to the MTD. The rest of the XML schema specification for the taskflow environment is listed in Appendix B.

Figure 3.19 shows an example of a multi-task definition layer written in XML. The name attribute for MultiTaskDefn has a value of task.parabola. The list of
input and output data ports and their types are specified using the XML tags Input and Output and XML attributes port and type. The TaskList element gives a list of tasks, their instance names and their corresponding task references.

As such, these XML files can be directly written by hand in construction of taskflows. However, the preferred choice is to use an XML editor, and the reason is clear once we examine the interface as shown in Figure 3.20. The entry of the taskflow configuration is well-structured, moreover, we get the benefit of the validating editor. Figure 3.20 shows a screenshot of one such XML editor that is being used for construction of task_parabola taskflow described earlier. It shows that the task_parabola contains:

- (1) five input data ports named emailaddr, minDescrA, minDescrB, minDescrC, and minInitCost,
- one output data port named mOutReport,
<MultiTaskDefn name="task_parabola" bodyRef="parabola_body.cdt#task_parabola">
  <Title> Taskflow task_parabola </Title>
  <InputList>
    <Input port="emailaddr" type="temporary">
      <Title> Email address </Title>
    </Input>
    <Input port="mInDescrA" type="persistent">
      <Title> Input description for task A </Title>
    </Input>
    <Input port="mInDescrB" type="persistent">
      <Title> Input description for task B </Title>
    </Input>
    <Input port="mInDescrC" type="persistent">
      <Title> Input description for task C </Title>
    </Input>
    <Input port="mInitCost" type="temporary">
      <Title> Initial cost (should be very large) </Title>
    </Input>
  </InputList>
  <OutputList>
    <Output port="mOutReport" type="persistent">
      <Title> Output report of parabolic evaluation </Title>
    </Output>
  </OutputList>
  <TaskList>
    <Begin/>
    <Task instance="(InitA)" taskRef="parabola_defn.cdt#InitSolver"/>
    <Task instance="(InitB)" taskRef="parabola_defn.cdt#InitSolver"/>
    <Task instance="(InitC)" taskRef="parabola_defn.cdt#InitSolver"/>
    <Task instance="(A)" taskRef="parabola_defn.cdt#SolverFlow"/>
    <Task instance="(B)" taskRef="parabola_defn.cdt#SolverFlow"/>
    <Task instance="(C)" taskRef="parabola_defn.cdt#SolverFlow"/>
    <Task instance="(D)" taskRef="parabola_defn.cdt#Evaluator"/>
    <Task instance="(E)" taskRef="parabola_defn.cdt#Report"/>
    <End/>
  </TaskList>
  <TaskGraph>
    (BEGIN)  =>  (InitA)  =>  (A)  =>  (D)  =>  (E)  =>  (END)  (A)  =>  (A)
    (BEGIN)  =>  (InitB)  =>  (B)  =>  (D)  =>  (E)  =>  (END)  (B)  =>  (B)
    (BEGIN)  =>  (InitC)  =>  (C)  =>  (D)  =>  (E)  =>  (END)  (C)  =>  (C)
  </TaskGraph>
</MultiTaskDefn>

Figure 3.19: An example for multi-task definition written in XML.
Figure 3.20: Screenshot of taskflow parabola in an XML editor.

- eight task instances named (InitA), (InitB), (InitC), (A), (B), (C), (D), and (E), and
- task graph with control-edges specifying the dependencies among the eight task instances of the parabola flow.

In addition, it also lists five single-task definitions and one more hierarchical multi-task definition that are referenced by taskflow task_parabola as one of its subtasks.

## 3.5 Single-User Implementation and GUI

In this section, we look at one important aspect of the taskflows that deals with the user interface of the OmniFlow environment. We first introduce the various concepts of graphical user interface (GUI) design below that leads to a simple yet
highly interactive user interface for the whitebox component of the taskflow. This is followed by a discussion on the reference implementation of its GUI.

3.5.1 Conceptual GUI Design: A Tree View and a Graph View

There are several elements in a whitebox component of a taskflow and it is difficult to represent all the elements in a taskflow as a single view without making it overly complex and hard to understand to the user. We therefore present two different views of the same taskflow component to the user: (1) a hierarchical tree view of all the taskflow components, and (2) a graph view rendering of any multi-taskflow at a given level of hierarchy. Both the views provide the full functionality of selection, control and execution of the various tasks in the OmniFlow environment and these views differ in only the way the user is presented information about the task. We next discuss both the views in detail:

Tree view. This consists of the main task instance displayed as the root of the tree view. The children of the main task instance can be opened or closed by the user by clicking on a ‘+’ or a ‘-’ symbol located near the task instance node. On opening the main task instance, it displays the data dependencies, if any, of the main task, such as InputList, InOutList, OutInList and OutputList and also its TaskList.

Figure 3.21 (a) shows a conceptual tree view of one such taskflow. It consists of two input data ports mIn1 and mIn2 and one output data port mOut1. The values for each data port are initialized by reading the corresponding values stored in the cdtML files, however, these values can be interactively changed by the user before invocation of a task. Under TaskList, it shows three tasks (TaskA), (TaskB), (TaskC), and one (Begin) and one (End) node. Here, (TaskA) and (TaskC) are multi-tasks as these contain TaskList as its children in addition to InputList and OutputList, whereas (TaskB) is a single-task since it only contains InputList and OutputList as its children. Since only multi-tasks have a corresponding graphical representation, multi-tasks have a LoadGraphView button located next to it which can be used to load the graph view.
The tree view displays the hierarchical view of the task instances in the OmniFlow environment. It provides an intuitive user interface to the user to interactively browse through the entire hierarchy by opening or closing the appropriate task instances. The task instance names, displayed in the browser view, are themselves buttons so that users may click on it to invoke and execute the corresponding task.

The graph view provides the user with an interface that readily depicts the various task-to-task, data-to-task and task-to-data relationships. Users may click on the control-edges to select or de-select the execution of a part of the taskflow. During execution, it displays the current state of each task using unique colors.

Figure 3.21: Conceptual taskflow GUI design: a tree view and a graph view.

The name of the task instance, displayed in the tree view, is itself a button widget. Users can click on the task button to invoke the execution of the corresponding task. Once the task is executing the Abort button corresponding to that task becomes active so that users may click on it if they decide to abort the task. On the other hand, an additional button Clean is provided to initialize or delete the output files before task invocation.

In general, the tree view provides a simple, compact user-interface for browsing the hierarchy of the task instances as well as for its interactive execution. However,
it is difficult to readily depict the various task-to-task, data-to-task and task-to-data dependency relationships in the OmniFlow environment. We have therefore also developed a graph view for the OmniFlow environment which is discussed next.

**Graph View.** At any given level of hierarchy, the task instances of a multi-task can be represented as a graph consisting of: (1) control-edges depicting task-to-task dependencies, corresponding to the TaskGraph, and (2) data-edges depicting data-to-task and task-to-data dependencies, corresponding to the input, inout, outin and output dataflow for the task instance.

Figure 3.21 (b) shows the earlier example of the tree view, rendered as a graph. Each task instance name is displayed as an button widget, which can be invoked for execution. It also consists of the following additional elements:

- a list of input, inout and outin data to the task instance, represented as circles at the top, and connected with directed data-edges to the task instance,

- a **skip** checkbox that can be selected if the user wants to skip the execution of the task instance,

- an **exec** checkbox that can be selected if the user wants to force the execution of the task instance without checking for the timestamps of the input/output data dependencies,

- a **LGV** button for loading the graph view if the task instance corresponds to a multi-task,

- an **Abort** button, which becomes active only when the corresponding task instance is executing,

- a **Clean** button, that can be used to delete the output files for the task instance before invocation,

- a list of inout, outin, and output data to the task instance, represented as circles at the bottom, and connected with directed data-edges from the task instance.
All the elements of the task instance are encapsulated in a shaded box. The edge of the encapsulated box is displayed thick if the task instance is a multi-task, otherwise it is displayed thin for a single-task.

The various task instances are connected by bold directed control-edges that can be opened or closed by the user by clicking on it. A square box is shown at the head of each control-edge and it corresponds to the output status ports of the ControlFork primitive. During execution, the current state of each task instance is displayed as a unique color in the output status port of the task instance. The color scheme used to indicate the different states of a task instance is shown in Figure 3.21 (b) on the bottom right.

A graph view is very useful in providing a visual representation of the various dependency relationships in a taskflow. However, the main limitation of the graph view is that it displays only one level of hierarchy of a task instance on a single view. Thus, rather than opening multiple graph views for each level of hierarchy, in general, it is more convenient to use a tree view to browse the taskflow hierarchy and selectively load the graph views of only few task instances that are of interest to the user.

### 3.5.2 GUI Implementation and Taskflow Execution Engine

The implementation of the GUI and its execution engine can be divided into two parts: (1) initial loading of the taskflow files, and (2) scheduling the task instances for execution during run-time. Due to the nature of the problem, it is important to have an efficient implementation of the scheduling algorithm because it constitutes a major part during execution as compared to loading of the taskflow. The scalability experiment in Chapter 5 measures the performance characteristics of the implementation prototype. Figure 3.22 shows the implementation architecture, which we next discuss below:

**Taskflow Loader.** The taskflow loader consists of three stages: (1) Tcl-xml parser to read the cdtML specification files for taskflows, written in xml, (2) taskflow specs validator to verify the correctness of the xml files against cdtML schema, and (3)
The flow on the left corresponds to the initial reading and parsing stage of the implementation whereas the flow on the right (shown in shaded box) corresponds to the taskflow scheduling algorithm during execution. Task instances are dynamically loaded and scheduled for execution during run-time.

Figure 3.22: Implementation architecture of taskflow scheduling engine.

display tree/graph view to render the xml files in a GUI.

On invocation, the loader merely reads the main invocation part of taskflow and renders it in the GUI. The loading of the rest of the taskflow specification files in xml is delayed until it is actually required due to: (1) a user request to load/view the task instance in the GUI, or (2) the task instance needs to be invoked for execution. This results in better loading time of the taskflow instances.

**Taskflow Scheduler.** The taskflow scheduler consists of a recursive algorithm which calls itself for every occurrence of multi-task instances in the taskflow. When a user invokes a task instance for execution, the taskflow scheduler first evaluates the ControlJoin condition and accordingly changes the state of the task to Enabled mode. Once enabled and if it is a multi-task instance, it immediately changes to Executing state, expands the multi-task instance into another level of hierarchy, and schedules the (BEGIN) node for execution. On the other hand, if it is a single-task instance, it waits until the corresponding host becomes idle, and then schedules the task for execution.

Upon completion of the task execution, it evaluates the RepeatCondition for the task instance, if any, to re-invoke the task several times until the condition is satisfied.
After that, it evaluates the **ControlFork** condition to validate the status of each of its output control-edge. For each of the successor task of the task instance, it invokes the taskflow scheduler for evaluation of its **ControlJoin** condition and execution.

### 3.5.3 Example of Concurrent Execution

Here, we show an implementation of the ‘parabola’ taskflow that has been discussed earlier in Section 3.1 in Figure 3.4. The implementation of the parabola taskflow consists of a total of fifteen task instances which refers to two multi-task definitions and five single-task definitions. Figure 3.23 shows the GUI representation of the toplevel multi-task instance and the entire cdtML specification files for the parabola taskflow, which is written in xml, is given in Appendix D.

As discussed previously in the conceptual GUI design, the actual graph representation of the parabola taskflow consists of the multi-task instances (A), (B) and (C) are displayed with bold edges whereas the remaining single-task instances are displayed with thin edges. Figure 3.23 also shows the expanded view of the multi-task instance (A) which consists of two single-task instances (**pSolverWeb**) and (**download**). In addition, each of the multi-task instance is repeated several times as shown by the feedback edge.

The bottom part of Figure 3.23 shows an execution trace of invoking the ‘parabola’ taskflow. Due to lack of space, only part of execution trace is shown here. Examining lines 16 through 25, it is evident that it schedules the three multi-task instances (A) (B) and (C) simultaneously for concurrent execution. Furthermore, lines 38 through 42 show that task instance (C) is invoked for repeated execution. Finally, lines 169 through 170 show that task instance (D) is not enabled until all of its previous three tasks have completed execution. The results of executing the **parabola** taskflow is shown on the last line, where it reports a total cost of -104.0 and average cost of -34.67 for execution of three tasks.
Figure 3.23: Concurrent execution in a taskflow: taskA, taskB, and taskC.
3.6 Evaluation of Taskflow Patterns

The task model described earlier can be used to construct very simple to large complex taskflows. The ControlJoin and ControlFork tasks, along with encapsulated blackbox components and whitebox components, used in construction of taskflows results in graphs that represent several patterns including sequencing, splits, concurrency, joins, iterations and cycles. The functionality of the taskflow environment largely depends on how well it handles the different kinds of patterns occurring in a taskflow. The various patterns occurring in a typical workflow have been identified, evaluated and reported in Advanced Workflow Patterns by W.M.P. van der Aalst et al. in Proceedings Seventh IFCIS International Conference on Cooperative Information Systems, September 2000 [110]. We make use of the ten patterns described in [110] to evaluate the functionality of the OmniFlow environment.

Figure 3.24 provides a comparative summary of what level of support is offered by the various environments evaluated in [110]. The table also reports results of evaluating the OmniFlow environment for these patterns. Overall, it is found that MQSeries/Workflow supports a direct implementation of five out of ten patterns, whereas all other environments support less. As for the OmniFlow environment, it support seven out of ten patterns listed in the table. In addition, the OmniFlow environment also supports two more patterns that are not listed in Figure 3.24, such as recursion and manual synchronization.

We next describe several representative taskflow patterns in detail and explain with an execution trace of simulating the OmniFlow environment that resolves these patterns. The execution trace displays the state and the name of each task in the hierarchy. Additionally, when each task goes into ‘done’ state, it also displays the amount of time taken to execute the task. To simulate the various taskflow patterns, we have used a sleep task which generates a random number from 0 to 5000 milliseconds for use as argument to the sleep command. The amount of time that a task sleeps for in a simulation environment is also shown in the execution trace when the task goes into ‘executing’ state. In certain cases, we have introduced a fixed amount of sleep time to simulate the taskflow pattern correctly.
<table>
<thead>
<tr>
<th>Workflow Pattern</th>
<th>Level of Support Environments evaluated in [110]</th>
<th>OmniFlow Env.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Synchronizing merge</td>
<td>MQSeries/Workflow and Inconcort support merge, others implement using X-OR split constructs</td>
<td>yes, ControlJoin construct directly supports merge</td>
</tr>
<tr>
<td>L-out-of-M join</td>
<td>Verve uses 1-out-of-M join (discriminator) combined with AND join and AND splits, hard to implement in others</td>
<td>yes, ControlJoin construct directly supports L-out-of-M join</td>
</tr>
<tr>
<td>Arbitrary cycles</td>
<td>MQSeries/Workflow, Inconcort has decomposition construct, Visual WorkFlo, SAP R/3 has special loop construct</td>
<td>partial, arbitrary cycles need to be transformed into structured cycles</td>
</tr>
<tr>
<td>Implicit termination</td>
<td>Staffware, MQSeries/Workflow, Inconcort terminate processes when idle, others allow only single exit node</td>
<td>allows only single exit node, so implicit termination does not occur</td>
</tr>
<tr>
<td>Multiple Instances (apriori knowledge)</td>
<td>MQSeries/Workflow provides special ‘Bundle’ construct to instantiate number of instances, others do it sequentially</td>
<td>yes, use structured cycles</td>
</tr>
<tr>
<td>Multiple Instances (no apriori knowledge)</td>
<td>Forte, Verve use loop and parallel split, Visual WorkFlo supports Release, I-Flow supports Chained Process Node</td>
<td>yes, use structured cycles</td>
</tr>
<tr>
<td>Multiple Instances requiring synchronization</td>
<td>MQSeries/Workflow’s ‘Bundle’ construct allows synchronization of created instances, not easy to implement in others</td>
<td>partial, limited to simple join conditions</td>
</tr>
<tr>
<td>Deferred choice</td>
<td>cancel other choice when selected for execution, or add an extra task to implement implicit-OR using explicit-OR construct</td>
<td>yes, ControlJoin construct directly supports deferred choice</td>
</tr>
<tr>
<td>Interleaved routing</td>
<td>pre-define sequence, or use deferred XOR-split construct, or for petri-net based models, add extra place as input/output of concurrent activities</td>
<td>no, this pattern is not directly supported</td>
</tr>
<tr>
<td>Milestone</td>
<td>introduce a boolean variable and check its value periodically</td>
<td>yes, use ControlJoin construct</td>
</tr>
<tr>
<td>Overall</td>
<td>MQSeries/Workflow supports a direct implementation of 5/10 patterns, all others support less</td>
<td>OmniFlow environment supports a direct implementation of 7/10 patterns.</td>
</tr>
</tbody>
</table>

Figure 3.24: Summary of ten advanced workflow patterns.
3.6.1 Synchronizing Merge

A synchronizing merge pattern involves two or more tasks that can be executed concurrently and a common task following it. However, there are several cases that are likely to occur when a number of tasks can be executed concurrently.

Consider three tasks A, B, and C as shown in Figure 3.25, where tasks A and task B can be executed concurrently and C is the common successor task for both A and B. Typical applications require that task C should not be invoked unless all of its predecessor tasks, namely A and B, have completed execution. However, many times only few concurrent tasks need to be invoked. This can happen, for example, when tasks A and B represent two different algorithms, task C represents a summary generator, and tasks A and B may not be necessarily available for execution at all times.

Thus, the occurrence of a synchronizing merge pattern in this example represents the following three cases:

1. When both the algorithms are available for execution, the summary task not should be invoked when only one of the algorithm completes execution, but its should wait for both the algorithms to complete execution.

2. When only one of the two algorithms is available for execution, we would still like to execute that algorithm and generate a corresponding summary. In such a case, the summary task should be invoked as soon as the first algorithm completes execution and it should not wait for completion of the other algorithm since it will never be executed.

3. When none of the algorithms are available for execution, the summary task should not be invoked be invoked at

A simple AND join of task A and B may be sufficient to resolve the first and third cases, but it will fail for the second case because task B, which is not invoked, will prevent the task C from executing even after task A completes. On the other hand, changing it to a simple OR join of task A and B would resolve the second case, but fail for the first case.
The execution trace represents three different cases of synchronize merge pattern: (1) both tasks A and B are invoked and executed, (2) only task A is invoked while task B is skipped, and (3) both tasks A and B are skipped.

<table>
<thead>
<tr>
<th>UserInvokeTask: (SyMrg) (BEGIN)</th>
<th>20 EnabledTask = (SyMrg) (B)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BeginForkTask = (SyMrg) (BEGIN)</td>
<td>21 SkippedTask = (SyMrg) (B)</td>
</tr>
<tr>
<td>2 EnabledTask = (SyMrg) (A)</td>
<td>22 ExecutingTask = (SyMrg) (A)</td>
</tr>
<tr>
<td>3 EnabledTask = (SyMrg) (B)</td>
<td>23 Sleep (4109 ms) = (SyMrg) (A)</td>
</tr>
<tr>
<td>4 ExecutingTask = (SyMrg) (A)</td>
<td>24 CompletedTask = (SyMrg) (A)</td>
</tr>
<tr>
<td>5 Sleep (4390 ms) = (SyMrg) (A)</td>
<td>25 DoneTask = (SyMrg) (A) 4146 ms</td>
</tr>
<tr>
<td>6 ExecutingTask = (SyMrg) (B)</td>
<td>26 EnabledTask = (SyMrg) (C)</td>
</tr>
<tr>
<td>7 Sleep (2904 ms) = (SyMrg) (B)</td>
<td>27 ExecutingTask = (SyMrg) (C)</td>
</tr>
<tr>
<td>8 CompletedTask = (SyMrg) (B) 2967 ms</td>
<td>28 Sleep (2632 ms) = (SyMrg) (C)</td>
</tr>
<tr>
<td>9 DoneTask = (SyMrg) (A) 4431 ms</td>
<td>30 DoneTask = (SyMrg) (C) 2869 ms</td>
</tr>
<tr>
<td>10 EnabledTask = (SyMrg) (C)</td>
<td>31 DoneTask = (SyMrg) (END) 7254 ms</td>
</tr>
<tr>
<td>11 ExecutingTask = (SyMrg) (C)</td>
<td>UserInvokeTask: (SyMrg) (BEGIN)</td>
</tr>
<tr>
<td>12 ExecutingTask = (SyMrg) (C)</td>
<td>32 BeginForkTask = (SyMrg) (BEGIN)</td>
</tr>
<tr>
<td>13 ExecutingTask = (SyMrg) (C)</td>
<td>33 EnabledTask = (SyMrg) (A)</td>
</tr>
<tr>
<td>14 ExecutingTask = (SyMrg) (C)</td>
<td>34 SkippedTask = (SyMrg) (A)</td>
</tr>
<tr>
<td>15 CompletedTask = (SyMrg) (C)</td>
<td>35 EnabledTask = (SyMrg) (B)</td>
</tr>
<tr>
<td>16 DoneTask = (SyMrg) (C) 4467 ms</td>
<td>36 SkippedTask = (SyMrg) (B)</td>
</tr>
<tr>
<td>17 DoneTask = (SyMrg) (END) 9343 ms</td>
<td>37 SkippedTask = (SyMrg) (C)</td>
</tr>
<tr>
<td>18 UserInvokeTask: (SyMrg) (BEGIN)</td>
<td>38 SkippedTask = (SyMrg) (END)</td>
</tr>
<tr>
<td>19 BeginForkTask = (SyMrg) (BEGIN)</td>
<td>39 DoneTask = (SyMrg) (END) 185 ms</td>
</tr>
</tbody>
</table>

Figure 3.25: Taskflow pattern: synchronized merge.

This problem is overcome by specifying a combination of AND/OR join conditions for task C such that it makes use of the state of task A and task B to invoke correctly. The join condition for the current example is \( OR( AND(A \text{ valid}, B \text{ valid}), \ AND(A \text{ valid}, B \text{ skip}), \ AND(A \text{ skip}, B \text{ valid}) ) \)

The execution trace of simulating the synchronizing merge pattern is shown in Figure 3.25. It consists of three user invocations, corresponding to the three different cases described above.

1. For the first case, tasks A and B are enabled and start executing concurrently, as shown in line 4 and line 6. Since both the tasks are invoked, task C is not enabled until both have completed, which occurs on line 12.

2. For the second case, task A starts executing on line 22 whereas task B is skips
execution on line 21. Therefore, task C is invoked as soon as task A completes on line 26.

3. For the third case, both tasks A and B skip execution on line 34 and line 36 respectively. In this case, task C is not enabled and it also skips execution, as shown in line 37.

3.6.2 Synchronizing L-out-of-M Join and Abort

The synchronizing L-out-of-M join pattern consists of M concurrent tasks, out of which only L tasks are necessary to be completed to invoke the common successor task. Once the minimum number of tasks have completed, we should abort the remaining concurrent tasks which are still executing.

Consider the task of obtaining an airfare quote from a number of on-line travel agents, say five. It may be sufficient to make a purchase decision as soon as the first three quotes are available from any of the five travel agents and abort the request sent to remaining travel agents.

Figure 3.26 shows two examples, both requiring 2-out-of-3 join, but the only one aborts the remaining tasks. Each consists of three concurrent tasks A, B and C and task D has the specified 2-out-of-3 join. However, the example on the right also has abort edges, shown as shaded edges, from task D to tasks A, B and C. These edges send an abort signal to all the tasks as soon as task D is enabled for execution.

The execution trace for the simulation of these patterns is shown below each example in Figure 3.26. For 2-out-of-3 join, the trace shows that tasks A, B and C are executed concurrently on line 5, 7 and 9. As soon as task A completes on line 14 and task B on line 12, task D is immediately enabled for execution, without waiting for task C to complete. Moreover, in this simulation, task D completes execution on line 19 even before task C completes execution on line 22. The second execution trace is similar to the first one, except for the difference that task C immediately aborts on line 16 as soon as task D is enabled on line 15.
The execution trace shows that task D is invoked as soon as tasks A and B have completed and without waiting for task C to complete.

The execution trace shows that task D is not only invoked as soon as tasks A and B, without waiting for task C, but it also aborts task C.

Figure 3.26: Taskflow pattern: synchronize L-out-of-M join.

### 3.6.3 Synchronizing Milestone

A synchronizing milestone pattern is a special taskflow pattern where a certain task can be invoked only as long as some other task has not completed execution.
A simple example for a milestone pattern is the request to expedite the shipment of a previously purchased item in an online store. The shipment of a purchased item can be expedited only as long as the item has not been shipped.

The execution trace shows two cases of milestone pattern: (1) task D is skipped because task C is already completed by the time D is enabled, and (2) task D is invoked because task C has not yet completed when D is enabled.

UserInvokeTask: (Ms) (BEGIN)
1 BeginForkTask = (Ms) (BEGIN)
2 EnabledTask = (Ms) (A)
3 EnabledTask = (Ms) (B)
4 ExecutingTask = (Ms) (A)
5 Sleep (4048 ms) = (Ms) (A)
6 ExecutingTask = (Ms) (B)
7 Sleep (6000 ms) = (Ms) (B)
8 CompletedTask = (Ms) (A) 4094 ms
9 DoneTask = (Ms) (A)
10 EnabledTask = (Ms) (C)
11 ExecutingTask = (Ms) (C)
12 Sleep (142 ms) = (Ms) (C)
13 CompletedTask = (Ms) (C) 182 ms
14 DoneTask = (Ms) (C)
15 CompletedTask = (Ms) (B) 6039 ms
16 DoneTask = (Ms) (B)
17 SkippedTask = (Ms) (D)
18 SkippedTask = (Ms) (END)
19 DoneTask = (Ms) (END) 6278 ms
UserInvokeTask: (Ms) (BEGIN)
20 BeginForkTask = (Ms) (BEGIN)
21 EnabledTask = (Ms) (A)
22 EnabledTask = (Ms) (B)
23 ExecutingTask = (Ms) (A)
24 Sleep (3402 ms) = (Ms) (A)
25 ExecutingTask = (Ms) (B)
26 Sleep (6000 ms) = (Ms) (B)
27 CompletedTask = (Ms) (A)
28 DoneTask = (Ms) (A) 3438 ms
29 EnabledTask = (Ms) (C)
30 ExecutingTask = (Ms) (C)
31 Sleep (4456 ms) = (Ms) (C)
32 CompletedTask = (Ms) (B)
33 DoneTask = (Ms) (B) 6036 ms
34 EnabledTask = (Ms) (D)
35 ExecutingTask = (Ms) (D)
36 Sleep (2567 ms) = (Ms) (D)
37 CompletedTask = (Ms) (C)
38 DoneTask = (Ms) (C) 4492 ms
39 CompletedTask = (Ms) (D)
40 DoneTask = (Ms) (D) 2001 ms
41 DoneTask = (Ms) (END) 8889 ms

Figure 3.27: Taskflow pattern: milestone.

In Figure 3.27, task A represents the purchase order for an item, task B represents the request for expediting the shipment, task C represents the shipment of an item and task D represents processing the expediting request.

This problem is resolved by use of the NOT function in specification of the ControlJoin condition of task D. The NOT condition is also represented as a white circle in the figure at the tail of the edge between task C and task D. This condition prevents task D from executing if task C has already completed execution.

The execution trace of the milestone pattern shows the simulation of two cases: (1)
task D skips the execution on line 17 because task C has already completed execution on line 14, (2) task D is enabled for execution on line 34 because task B completes (line 33) while task C is still executing which can be inferred from lines 30, 31 and 38.

### 3.6.4 Deferred Choice

Deferred choice pattern represents a task where a number of concurrent tasks are enabled for execution, however only one needs to be executed. This can occur when it is possible to execute the same task on a number of resources, but only one task needs to be executed depending on which resource is available for processing.

![Diagram](image.png)

The execution trace shows that both tasks A and B are enabled simultaneously, but as soon as task A starts executing, task B is aborted.

<table>
<thead>
<tr>
<th>UserInvokeTask: (Dc) (BEGIN)</th>
<th>8 DoneTask = (Dc) (A) 1863 ms</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 BeginForkTask = (Dc) (BEGIN)</td>
<td>9 EnabledTask = (Dc) (C)</td>
</tr>
<tr>
<td>2 EnabledTask = (Dc) (A)</td>
<td>10 ExecutingTask = (Dc) (C)</td>
</tr>
<tr>
<td>3 EnabledTask = (Dc) (B)</td>
<td>11 Sleep (5230 ms) = (Dc) (C)</td>
</tr>
<tr>
<td>4 ExecutingTask = (Dc) (A)</td>
<td>12 CompletedTask = (Dc) (C)</td>
</tr>
<tr>
<td>5 AbortedTask = (Dc) (B)</td>
<td>13 DoneTask = (Dc) (C) 3274 ms</td>
</tr>
<tr>
<td>6 Sleep (1785 ms) = (Dc) (A)</td>
<td>14 DoneTask = (Dc) (END) 5428 ms</td>
</tr>
<tr>
<td>7 CompletedTask = (Dc) (A)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.28: Taskflow pattern: deferred choice.

In Figure 3.28, both the tasks A and B are instances of the same task except for the difference that they are invoked on different hosts. In addition, both the tasks also have abort edges connected to each other, represented as shaded lines, which are used to abort the other task as soon as one starts executing.

The execution trace for simulating the deferred choice pattern is shown in Figure 3.28. Here, both the tasks A and B are enabled for execution on lines 2 and 3 respectively. On line 4, task A begins execution which in turn sends an abort signal to task B, which is then aborted on line 5.
3.6.5 Recursion

A recursion pattern occurs in a taskflow if it contains a hierarchical task, which when expanded into its sub-taskflow, encapsulates itself as one of its sub-tasks.

![Diagram of recursion taskflow]

The execution trace shows that task C is recursively invoked three times before the terminating condition is met, at which point, the task C completely comes out of the recursion loop.

```plaintext
# Taskflow pattern: recursion.
```

Figure 3.29: Taskflow pattern: recursion.

An example of a recursive task is shown in Figure 3.29. C is a hierarchical task, which consists of two sub-tasks: itself task C and another task B. Such a taskflow also has potential of getting into an infinite loop of recursive calls. Therefore, it is important to have a terminating condition, which will eventually stop the recursive invocation of the same task.
The execution trace for such a recursive task is shown in Figure 3.29. The task C
is recursively invoked, first at level 1 on line 10 and next at level 2 on line 21. However,
the taskflow terminates at level 3 of recursive call because at this level the task B
generates an output data that satisfies a user specified criteria, thereby resulting in
task C being skipped at level 3. After that the task C completely comes out of the
recursion loop, one level at a time.

3.6.6 Structured Cycles

Most taskflows contain cycles because a sequence of tasks need to iterated several
times before a task can be completed. Structured cycles pattern in a taskflow allow
the same task to be repeated several times until an exit criteria is satisfied. On the
other hand, arbitrary cycles in a taskflow can lead to potential deadlock problems,
hence only structured cycles are allowed.

Figure 3.30 shows a taskflow consisting of two tasks A and B, out of which task B
contains a feedback edge to repeatedly invokes task B. For example, task A may be
representing a circuit synthesis tool which needs to be repeatedly invoked different
parameters until the synthesized circuit meets the design constraints.

The execution trace for the simulation of this example shows that task A is invoked
once on lines 2 through 6 and task B is invoked first time on lines 7 through 10. After
that task B is repeatedly invoked two more times, on lines 12 through 15 and on lines
17 through 20. On line 21, task B is no longer repeated because the exit criteria,
which was specified as a maximum limit of three repetitions, has been satisfied.

3.6.7 Other Taskflow patterns

Several other taskflow patterns include arbitrary cycles, implicit termination, and
dynamic instances of a task with or without apriori runtime knowledge.
The execution trace shows that task B is repeated invoked three times before the terminating condition is met, at which point, the task B stop repeating execution.

UserInvokeTask: (Cycle) (BEGIN)
1  BeginForkTask = (Cycle) (BEGIN)
2  EnabledTask = (Cycle) (A)
3  ExecutingTask = (Cycle) (A)
4  Sleep (610 ms) = (Cycle) (A)
5  CompletedTask = (Cycle) (A)
6  DoneTask = (Cycle) (A) 582 ms
7  EnabledTask = (Cycle) (B)
8  ExecutingTask = (Cycle) (B)
9  Sleep (1641 ms) = (Cycle) (B)
10 CompletedTask = (Cycle) (B)
11 RepeatTask = (Cycle) (B)
12 EnabledTask = (Cycle) (B)
13 ExecutingTask = (Cycle) (B)
14 Sleep (2973 ms) = (Cycle) (B)
15 CompletedTask = (Cycle) (B)
16 RepeatTask = (Cycle) (B)
17 EnabledTask = (Cycle) (B)
18 ExecutingTask = (Cycle) (B)
19 Sleep (741 ms) = (Cycle) (B)
20 CompletedTask = (Cycle) (B)
21 DoneTask = (Cycle) (B) 776 ms
22 DoneTask = (Cycle) (END) 6376 ms

Figure 3.30: Taskflow pattern: structured cycle.

Arbitrary cycles

As mentioned earlier, most taskflows contain cycles because a sequence of tasks need to iterated several times before a task can be completed. However, arbitrary cycles pattern can potentially lead to infinite loops and deadlock conditions, unless designed properly.

Arbitrary cycles such as one shown with tasks A, B, C and D can result in deadlock as well as infinite loops. To overcome this limitation, tasks B and C are replicated and two new hierarchical tasks are created. Each hierarchical task then uses a structured cycle to repeat as necessary.

Figure 3.31: Taskflow pattern: Arbitrary cycle.
Consider the example shown in Figure 3.31. It consists of four tasks A, B, C and D executed in sequence. In addition, there is a feedback control-edge connecting task C to task A and another feedback control-edge connecting task D to task B. Due to such a configuration, task A will never be invoked because it will indefinitely wait for a token to arrive from task C, whereas task C is not invoked unless task A is also invoked, thereby resulting in a deadlock condition. Yet another problem arises when task B is invoked directly by the user, it is possible that while executing the loop containing tasks B, C and D, it may unintentionally start executing the loop containing tasks A, B and C.

We therefore allow only structured cycles, and no arbitrary cycles. Arbitrary cycles in a taskflow need to be transformed by the user into structured cycles by replicating the instances of a task to form a hierarchical taskflow.

These problems can be avoided by creating two hierarchical taskflows where the two tasks B and C are replicated. Each hierarchical taskflow can then be repeated using structured cycles, which will be described in the next section. Arbitrary cycles are, therefore, not allowed in the OmniFlow environment and all the tasks in a taskflow form a directed acyclic graph (DAG).

**Implicit Termination**

Implicit termination pattern occurs when taskflow has certain tasks that result in early termination. This can result in termination of the taskflow while some other concurrent tasks are active.

Consider a flow consisting of several tasks, within which two tasks C and D both have a terminating condition. Nominally, it is difficult to determine which terminating nodes have completed when, so that the execution state of the current task can be changed to completed.

In the OmniFlow environment, we only allow a single entry point and a single exit point to prevent such problems. In the above example, the output control-edges of task C and task D should be connected to EndJoin which is similar to a control join element and waits for all incoming tasks to complete or terminate.
Dynamic Instances with Apriori Runtime Knowledge

It may be necessary to invoke several instances of the same task. This can happen when the number of dynamic instances of a task that are invoked is decided during runtime by the number of data sets that need to be processed. We use data-dependent structured cycles to create dynamic instances of the task which is then repeated as many times as required by the number of data sets to be processed.

Dynamic instances with no apriori runtime knowledge

In addition to invoking dynamic instances of a task during runtime, it may be possible that the number of instances of task invocation is not know prior to task execution. This can happen when the dynamic instances of the task are dependent on one another. For this pattern, we conditional data-dependent task repetitions that iterates a single task number of times until required condition on data set is satisfied.

Dynamic instances (with Apriori) requiring synchronization

Taskflows containing dynamic invocations of a single task would also require synchronizing join so that a subsequent task can be invoked. Since, the number of instances of a task is only known during runtime, it is not possible to specify arbitrary join conditions for synchronizing dynamic instances of a single task. Therefore in an OmniFlow environment, we only allow simple join conditions such as wait for $L$ out of $n$ task instances to complete.

3.7 Summary

This chapter introduces an architecture for a programmable taskflow-oriented computing environment, called OmniFlow environment, and its implementation. The design and development of taskflows in the OmniFlow environment allows one to encapsulate any tool that can be represented as a blackbox component. A blackbox component may represent a wide variety of tools, ranging from legacy applications
to newly written programs, a generic cgi-script accessible on a web server to highly customized application specific services, such as JavaCADD.

This environment defines and makes use of whitebox components, that support a highly interactive composition of well-defined distributed blackbox components. Additionally, user configurable control-edges and abort-edges provides high level of interactivity during run-time execution of a whitebox component.

We also evaluated the OmniFlow environment in the context of various advanced workflow patterns reported in [110]. The evaluation results indicate reasonably good support for the workflow patterns - the OmniFlow environment supports a direct implementation of 7/10 patterns whereas other existing environments support a direct implementation of at most 5/10 patterns.
Chapter 4

Web-based Collaborative Computing Environment

The need to reconfigure OmniFlow into asynchronous and synchronous modes gives rise to a novel architecture, consisting of a centralized *asynchronous group server (AGS)* that supports OmniFlow assignments, re-configuration, and execution scheduling, and a distributed *asynchronous group client (AGC)* that is a copy of the OmniFlow archived on AGS. Both the server and the client are written in Tcl, once only, regardless of the applications, libraries, and data sets used in the project.

In the asynchronous mode, the OmniFlow may be considered a single-user application that is waiting to be made collaborative in the synchronous mode. Two major client/server architectures that can render single-user applications collaborative have been developed to date: a centralized architecture and a replicated architecture. Each has a number of advantages and drawbacks. A hybrid architecture attempts to overcome the drawbacks of both.

This chapter is organized into five sections as follows:

*Introduction:* representative examples of four typical taskflow projects, their requirements and needs for collaboration and a description of basic OmniDesk architecture.

*Asynchronous collaborative environment:* issues of invoking a shared task instance or
data, an architecture for coordinating task and data locking and implementation
details.

*Synchronous collaborative environment:* issues of sharing a simple text widget, an
architecture for synchronous collaboration and implementation details.

*Web-based environment:* an enhancement to the safe-tcl plugin to render tk appli-
cations collaborative without compromising security.

*Toolkit evaluations:* the proposed toolkits are evaluated and compared with existing
collaborative systems.

## 4.1 Introduction

Computer supported cooperative work (CSCW) and groupware systems deal with
collaboration necessary to engage a group of people to achieve a common task or goal
and provides an interface to a shared environment. One of the classification schemes
for the groupware systems is based on time space matrix [41], and is shown in Figure
2.5 in chapter 2.

- *Same place, same time:* The most popular and conventional example is face-to-
face interaction where several people sit together at the same time at the same
place and perform a group activity.

- *Same place, different time:* Multi-user editor is an example which allows asyn-
chronous editing of a document by several people at different times.

- *Different place, same time:* Shared white-boards and chat tools are examples
of conference systems that enable people distributed among various places to
hold synchronous distributed meetings.

- *Different place, different time:* Newsgroups and e-mail are a good example of
asynchronous distributed interaction where people read and post messages to
a newsgroup at different times from a different place or use e-mail to interact
with one another.
Systems that attempt to support the latter two, synchronous as well as asynchronous
distributed interactions, are the most interesting and is an area of continued research.

4.1.1 Representative Project Descriptions

We first examine a set of examples - projects that are from four completely different
domains and would help us identify the core issues that are common to all of them.
A project can be considered to be consisting of a number of tasks that need to be
executed in order to complete the project and may often involve more than one person
in the project. Consider the following four projects and the typical tasks that are
involved therein:

1. Book authoring: The objective of this project is to write and deliver a book to
the publisher. The book may involve one or more authors and will certainly
contain several book-reviews before the book is finalized. Book reviews could be
in several form such as direct contact, anonymous web review, etc. The various
tasks of a typical project which uses latex, are listed below:

   • write/edit/compile preface
   • write/edit/compile section1/tables/figs/citations
   • write/edit/compile section2/tables/figs/citations
   • write/edit/compile section3/tables/figs/citations
   • global compile preface/sections/bibliography/index
   • book review (repeat tasks if required)
   • publish and deliver the book to customer

2. Software development: The goal of the software development project is to design
and implement a system that can be delivered as a final product to a customer
or merely as an implementation prototype to the peers. Any software project
typically involves several iterations of code edits followed typically by compila-
tion and testing. It may also involve more than one person on its development
or testing. The various tasks of a typical software project is shown below:
• write/edit/compile/test/debug component 1
• write/edit/compile/test/debug component 2
• write/edit/compile/test/debug component 3
• global compile/test/debug all components
• code review (repeat tasks if required)
• package and deliver the software to customer

3. 

**VLSI chip design:** The goal of a chip design project is to manufacture and ship a VLSI chip for a specific application. Such projects usually require a large team of designers, each working on a sub-component of the chip. The project itself consists of several phases from behavioral design to logic synthesis, from netlist partitioning and to logic optimization, from layouting to test verifications. Each design phase is repeated several times back and forth before the final chip is ready for shipment. A high level view of the complexity of the various tasks in VLSI design process is shown below:

• write/edit behavioral design specs
• synthesize structural specs (netlist)
• partition netlist
• optimize each netlist partition
• layout each netlist partition
• simulate and verify for performance (repeat tasks if required)
• deliver the design to customer

4. 

**Benchmarking algorithms:** The objectives of this project is to benchmark related algorithms and publish their results. This project requires one to download and install a number of algorithms and execute them to evaluate the performance and compare the results of each with one another. The various tasks in benchmarking algorithms, called treatments, are listed below:
• assemble equivalence classes of benchmarks
• identify tools (treatments) that process all benchmarks
• invoke evaluation of results generated by each tool
• invoke comparisons and review of results (repeat tasks if required)
• generate internal report (comprehensive)
• generate external report (filtered contents)
• deliver reports to customers

The objectives and the tasks of the above examples, while unique to their respective domains, nevertheless, have a number of attributes that are common to all of them:

• All the projects involve a sequence of tasks that need to be executed repeatedly. Moreover, most of the tasks may rely on one or more tools that are distributed over the network. For example, VLSI design projects today typically rely on usage of several heterogeneous tools from different vendors that may be installed on dedicated servers. The OmniFlow environment, as discussed in the previous chapter, provides a very convenient mechanism to seamlessly access and execute these tools and data over the network.

• Most of the projects either directly involve a large group of people, as in VLSI design and software development, or may involve one or more persons at some instance. This requires the group of people to co-ordinate among themselves in an efficient manner to get the work done. It is important that the software developers can deliver a stable code to the testing community, while at the same time work on further development of their code. On the other hand, the testing community would have to provide appropriate feedback to the developers. This brings up issues such as how to share the data as well as well as other components effectively.

• Very often, there is need for a group of people, working on a related project, to interact with one another to discuss or resolve an issue. For example, in VLSI
design process, it is common for code designers to work with the layouting team to resolve a timing conflict of a critical path or simply to improve the circuit performance. Therefore, synchronous collaboration is a must under such conditions.

We next look at a generic abstraction of the above projects and identify the key issues involved therein, in order to support effective collaboration among project participants.

4.1.2 Requirements for Collaboration of Distributed Users

We consider the project issues in two environments: asynchronous collaborative and synchronous collaborative.

An Environment for Asynchronous Collaboration

When several people work on the same project, the first problem is the appropriate assignments of various tasks to each participant. An ad hoc interaction of the two users with the various tasks may result in unanticipated behavior. Both the users may try to invoke the same tools at the same time, or the two users may not use the right data for execution of the task assigned to them. It is important to co-ordinate the activities of the two or more users effectively for successful completion of the projects.

An environment for asynchronous collaboration is created by a partitioning of the taskflow among a number of participants. The project leader (Fred) partitions the taskflow by assigning ownership of tasks and related data to users supporting the project. Each user is expected to work on the task independently of others. In such a scenario, the need to synchronize data among users arises periodically. Each user must be able to access data updates from other users, and is expected to provide data updates to other users. Additional utility tasks must be provided to support these tasks.
An Environment for Synchronous Collaboration

When users need to collaborate synchronously on a specific application, such as a shared file editor in document writing or a shared layouting tool in VLSI design, there are several issues that should be addressed by the system which will provide a shared environment. Several possibilities exist even for a simple text widget that needs to be rendered collaborative in a shared environment. For example, a single-user text widget can be rendered as any of the following, based on the needs of the collaborating users at a specific instance:

- fully synchronized text editing with one of the user editing the text while others are observing,
- all users simultaneously editing different parts of the text and at the same time also observing what others are editing,
- all users simultaneously editing different parts of the text but only occasionally observing what others are editing, and
- all users simultaneously editing in an individual text widget and at the same time also observing what others are editing.

It becomes very difficult to anticipate in advance a suitable configuration preference for every user and such a collaborative application might turn out to be user-unfriendly or confusing for a particular team. Thus, it is important to provide the users with sufficient flexibility to share an application for collaboration that can be configured to match their preferences based on their needs.

The requirements of collaboration activities that arise in a group project are highlighted here. In particular, the partitioning of the taskflow into an asynchronous and a synchronous collaboration modes provides the basis for the two important client/server architectures, asynchronous client and group server (AGC/AGS) and synchronous client and group server (SGC/SGS). The OmniFlow environment and both of these architectures are integrated into a single OmniDesk environment, which is described in the next section.
4.1.3 Collaborative OmniDesk Environment

We propose a client/server based architecture, called OmniDesk, that addresses the issues of co-ordinating a project involving a team of distributed participants, distributed applications, libraries, and data sets and supports interactive collaboration among team participants. Accordingly, the OmniDesk architecture consists of two components: (1) an asynchronous group client/server architecture (AGC/AGS), and (2) a synchronous group client/server architecture (SGC/SGS), which are integrated with the OmniFlow environment to support collaboration activities among team participants.

Consider a collaborative project that consists of several tasks, \( T_1, T_2 \ldots T_k \), that are to be invoked and executed repeatedly by one or more team members in order to complete the project. Assuming there are up to \( a, b, \ldots n \) compute servers available for the project, each of the \( k \) tasks may be configured to be accessible on any one of the \( n \) compute servers.

Figure 4.1: OmniDesk architecture.

Figure 4.1 shows the basic architecture of the OmniDesk environment when there
are two participating team members: User1 and User2. Each user has an OmniFlow client installed on her computer system, which is represented as OFC1 and OFC2 in the architecture. The OmniFlow client allows both the users to access the tasks residing on n compute servers directly. However, it does not prevent the two users from invoking the same task or accessing the data for the task simultaneously. Therefore, to prevent potential conflicts arising due to simultaneous invocation of a task, we introduce an additional asynchronous group client, AGC1 and AGC2, for each user. The AGC1 and AGC2 clients enable communication between OFC1 and OFC2 via a centralized asynchronous group server, AGS. The OmniFlow clients, along with their respective AGC’s, always notify the AGS server when invoking any task or while accessing any task-data. Additionally, it also requests permission from AGS server prior to invocation of any of the task or accessing any task-data.

In addition to co-ordinating the various activities of the users for executing a task, it may be necessary for two users to interact directly with one another during runtime. We introduce two additional synchronous group clients, SGC1 and SGC2 respectively for each user, and a centralized synchronous group server, SGS, to share any information among the two users. SGS also maintains a library of inter-client synchronization (ICS) tables for each application. ICS tables define the default behavior of a shared application. For example, a text widget can be either configured as a shared file editor or as a chat window by using an appropriate ICS table.

Since SGS is centralized, it maintains consistent state information and therefore simplifies synchronization. SGS also maintains a log of events that have occurred. This is useful where users are allowed to join or leave a session arbitrarily at any time.

More details on the asynchronous and synchronous collaboration are discussed in the following sections.

4.2 Asynchronous Collaboration Environment

Consider the software development project whose goal is to design and implement a final deliverable product. Software projects typically involve several iterations of code edits, followed by compilation, testing and verification of test results. The code
development and the code-testing also usually involves more than one person. For the sake of simplicity, we consider here only four basic sequential tasks, A, B, C and D, that are to be assigned to two users, U1 and U2. The functionality of the four tasks is as follows:

**Task A** represents the editing of the code,

**Task B** represents the compilation of the code to generate an executable binary,

**Task C** represents invocation of the executable binary for testing, and

**Task D** represents invocation of a tool to verify the test results.

Because of the nature of the taskflow, it is natural to assign tasks A and B to one user, say Alice (shown as U1), and tasks C and D to another, say Bob (shown as U2). Such a taskflow is represented in Figure 4.2, which shows the task-to-task control edges and ownership assignments of each task to Alice and Bob. Additionally, it also shows a single data node that is common to both task B and task C and corresponds to the executable binary file for the software project. This data node presents a potential problem due to data sharing because the two tasks (B and C) accessing this data node are assigned to different users. Other data nodes, existing in the taskflow, are not shown, because those data nodes are automatically inherit the ownership of the corresponding task and do not present any problem.

![Figure 4.2: Issues with task ownership and data sharing.](image)

Typically, both Alice and Bob work independently on code-development and code-testing, respectively, by invoking the execution of the relevant tasks in the OmniFlow.
Potential conflicts can arise when Alice is executing task B, which compiles her code to create a new executable binary, and at the same time, Bob is trying to invoke the test program using the same executable binary. Simultaneous write/read access to such a shared binary file may result in corruption of data.

In general, there are three possibilities of accessing shared data by two tasks, but not all of these cases can result in corrupted data being read/written. Each case is explained briefly below:

**read/read access** If the tasks try to read the data simultaneously, there is no possibility of data corruption and hence there are no conflicts due to data sharing.

**read/write access** If the one task is trying to read the data, whereas the second task write new data at the same time, then this may result in incorrect data being read by the first task and hence a conflict occurs.

**write/write access** Finally, if both the tasks try to write simultaneously to the same data, this can also result in corrupted data and hence a conflict occurs.

We make use of read/write locks, stored at a centralized location using AGC/AGS client-server, to overcome such conflicts.

### 4.2.1 Architecture of Asynchronous Collaboration

The asynchronous collaboration architecture consists of two parts: (1) an asynchronous group server (AGS) that provides mechanisms for communication and maintenance of read/write locks for tasks and data in an OmniFlow, and (2) an asynchronous group client (AGC) that provides mechanisms for co-ordination of Omni-Flows among multiple clients.

The AGS server maintains two types of repositories: (1) *task-data ownership tables* - information about ownership of various tasks and data corresponding to an OmniFlow are maintained here; and (2) registered users and access permissions - only registered users are allowed access permissions to grab ownership of a task or data in an OmniFlow.
The AGC client is installed on each user’s machine and provides a mechanism for the OmniFlow client to communicate with the AGS server during runtime. Upon invocation, the AGC client establishes a socket connection with AGS and prompts the user to identify herself. Once the login process is completed, the user can query the status, grab and release ownership locks of a task or a data in an OmniFlow.

There are two types of ownership locks for data: (1) a read access lock, and (2) a write access lock. On the other hand, every task has a single type of ownership lock, called execution lock. The AGS server maintains the dynamic information about the read, write and execution ownership of the various tasks and task-data as the taskflow is being executed. The response of an AGS server when asked for a specific lock on a task or data depends on the current ownership state of that task or data.

Figure 4.3 (a) shows the table corresponding to a request for the read or write lock on a shared data. The shared data node is owned by two users and the table shows the case where the data has been locked by user U1 and user U2 is requesting to acquire a lock on the data. The AGS server responds to the request according to the current state of the lock acquired by user U1. When user U2 requests to acquire a read lock, the request is granted only if the state of U1’s lock is in read mode. For all other cases, it does not grant any lock to U2 until U1 releases her read or write lock on the shared data.

Similarly, Figure 4.3 (b) shows the table corresponding to a request for the execution lock of a task that is owned by two users. As shown, when a single task is owned by two or more users, each user has to send an invocation signal to the corresponding task so that it can be invoked.

Figure 4.3 (c) makes use of the software development project, discussed in the previous sections, to illustrate how the potential conflicts are resolved for shared task/data on an ownership boundary. The figures show two views of the OmniFlow, since there are two users, Alice and Bob, involved in this project. Both Alice and Bob get the same view, however they can interact with only those tasks that are owned by them. Thus Alice cannot invoke tasks C and D, whereas Bob cannot invoke tasks A and B. The shared data node is owned by both users, Alice and Bob. So, the AGC/AGS client-server prevents the data from getting corrupted by granting
(a) Shared data owner locking

<table>
<thead>
<tr>
<th>U1(status)</th>
<th>U2(acquire)</th>
<th>Lock?</th>
</tr>
</thead>
<tbody>
<tr>
<td>read</td>
<td>read</td>
<td>yes</td>
</tr>
<tr>
<td>read</td>
<td>write</td>
<td>no</td>
</tr>
<tr>
<td>write</td>
<td>read</td>
<td>no</td>
</tr>
<tr>
<td>write</td>
<td>write</td>
<td>no</td>
</tr>
</tbody>
</table>

(b) Shared task owner locking

<table>
<thead>
<tr>
<th>U1(status)</th>
<th>U2(status)</th>
<th>Invoke?</th>
</tr>
</thead>
<tbody>
<tr>
<td>idle</td>
<td>idle</td>
<td>no</td>
</tr>
<tr>
<td>idle</td>
<td>invoke</td>
<td>no</td>
</tr>
<tr>
<td>invoke</td>
<td>idle</td>
<td>no</td>
</tr>
<tr>
<td>invoke</td>
<td>invoke</td>
<td>yes</td>
</tr>
</tbody>
</table>

(c) Shared data ownership and locking

(d) Buffered task ownership and locking

Figure 4.3: Shared data and shared task ownership locking.
read/write access permissions according to Figure 4.3 (a).

Similarly, Figure 4.3 (d) overcomes the limitation of shared data conflicts by inserting an additional buffer task B-C, which is merely used to copy the data from its input to output whenever this task is invoked. Moreover, task B-C has been assigned to ownership to both Alice and Bob. This means that the task B-C cannot be invoked unless both users send an invocation signal to this task.

In the next section, we examine some of the implementation details of the asynchronous collaboration environment.

### 4.2.2 Implementation of AGS/AGC

The client-server architecture of the AGS/AGC toolkit is implemented using socket programming. Typically several users may be connected to AGS at any time. Each user needs to first establish a valid connection to the server by providing the login information. Figure 4.4 shows a snapshot of the login session that is typically used to connect to the group server. Users are required to provide an e-mail address and a login password to activate the connection. The hostname and the port number of the group server also needs to be provided. A registration window allows new users to create an account on the group server.

AGC users may invoke several taskflows and each taskflow can have different collaborative participants. As the number of users and the number of taskflows increase, this could lead to very complicated relationships. Also, it is important that the server as well as the client maintains this information in a clean fashion without mixing up any information with one another. Therefore, the asynchronous group server invokes a new safe interpreter, using the command `interp create -safe`, for every new client that has requested a socket connection and so also for every taskflow that is invoked. Similarly, the client also invokes a safe tcl interpreter for every taskflow that the user invokes, thereby providing reasonable security restrictions.

AGS and AGC communicate with one another using a well-defined set of APIs. There are basically two types of API commands: (1) synchronous API commands, which block the sequence of execution by waiting for the results; and (2) asynchronous
Registered users can connect to any group server by specifying its hostname and port number and by also providing an e-mail address and password on the top. The bottom part allows new users to register or existing users to change password.

Figure 4.4: Login session of a collaboration environment.

API commands, which do not block and wait for the results of the execution. When a client wants to acquire either a read lock or a write lock for a specific data or task instance, it is necessary to wait for the response from the AGS before invoking the task or accessing the data. In such a case, it is necessary to use synchronous API command that waits for the response.

On the other hand, when a client is releasing an acquired lock for a specific data or a task instance, it is sufficient to merely transmit the request to the AGS server and proceed with invocation of the remaining tasks, without having to wait for the response. Here, it is therefore possible to use an asynchronous API command.

However, the synchronous form of communication with the AGS server implies that these commands will block the socket channel between the client-server and prevent any other form of communication of using other commands. Therefore, we always send asynchronous commands over the socket channel, and make use of the ‘vwait’ command to implement the synchronous commands.

Accordingly, we have developed two simple procedures to handle these two types of commands, as shown below:
## 2. Synchronous transmission

```null
1. Asynchronous transmission

   # sid = socket id
   proc sendNforget {sid cmd} {
     transmitSocket $sid {} $cmd
   };
   # End of proc sendNforget

2. Synchronous transmission

   proc sendNreceive {sid cmd} {
     set returnId [clock clicks]
     transmitSocket $sid $returnId $cmd
     vwait $::returnId
   };
   # End of proc sendNreceive
```

The first procedure `sendNforget`, as the name suggests, sends the command across the socket channel and returns immediately. On the other hand, the second procedure `sendNreceive` transmits a unique variable name, generated using the clock command and waits for this variable to be set using the `vwait` command. When this data is received on the other side, the presence of a variable name implies that a result is expected and therefore it transmits back the results such that this unique variable name is set to contain the results.

These two procedures are used for implementing the communication of lock acquire and lock release procedures for the task instance and shared data with the group server.

### 4.2.3 Application Examples

Here we describe in brief three collaborative projects that required asynchronous participation from a team of two or more distributed members and illustrates the application of asynchronous collaboration in the context of each project. The complete set of three projects that used the asynchronous OmniFlowLite client is shown in Figure 4.5. The top half of the screenshots consist of the executable OmniFlowLite projects and the bottom half of the screenshots shows two file browsers: the left window for the files located on the server and the right window for the file located on the client side. It also shows the ownership of each file located on the server and the client.

A user may grab ownership of (1) a single file by simply double-clicking on the ownership display on the screen, (2) multiple files by selecting and highlighting a
(a) Configuration to execute a collaborative document composition

(b) Configuration to execute a collaborative Tcl/Tk compile

(c) Configuration to execute a collaborative experimental design

Figure 4.5: Three executable configurations of an asynchronous OmniFlowLite client.

range of files and clicking on the lock button located in the center of the two browser windows. Similarly, a user has the flexibility to release the ownership of a file by either directly double-clicking on a currently owned file or by highlighting a range of files and clicking on the release button in the center.

The three OmniFlowLite projects are summarized below:

(a) **Collaborative document composition** This flow consists of two or more authors collaboratively writing a common document using latex. It invokes five
tasks, \texttt{LaTeX}, \texttt{BibTeX}, \texttt{LaTeX Again}, \texttt{Dvi2ps} and \texttt{Ps2pdf}, necessary to execute in sequence in order to generate the final document in pdf format with required bibliographical references in it. Each task may be invoked by any of the participating authors, provided they have acquired ownership of the appropriate input text file documents. In Figure 4.5 (a), the client side of the file browser shows two authors, Brglez and Lavana. While the postscript file for Brglez is locked by ‘bgralez’, the postscript file for Lavana has been released ownership and hence not owned by anybody, denoted as ‘server’ under its file ownership.

(b) \textbf{Collaborative Tcl/Tk compile} The second project of two or more participants working on a complex Tcl/Tk application. Each may be writing and testing a set of procedures in isolation. The question is: will they work together as expected. By uploading the files from several sources to the server, a combined version can be assembled and compiled on the server, then accessed for execution among the participants. The Tcl/Tk compile flow consists of generating a master ‘pkgIndex.tcl’ file on the server, downloading all tcl/tk files developed by the various participating team members to the local client and invoking the tcl application on the local client.

(c) \textbf{Collaborative experimental design} This flow consists of creating a class of mutant circuits, based on the characterization of the set of reference circuits and uploading it to a central server for peer review. A team member may upload a set of reference circuit files to the server to create a mutant class of circuits. These are then characterized and laid out as schematics which can be accessed by another team for inspection and further analysis.

These flows have been implemented in OmniFlowLite [26] and are described in [28, 111, 112]. The OmniFlowLite implementation has also been demonstrated in the University Booth at the Design Automation Conference in June 2000 and a cross-platform source code executable posted on the Web [113]. The OpenWriter environment has been specified in [114] and the implementation status is presented in the senior project class report [115] and also discussed in Chapter 5.
4.3 Synchronous Collaboration Environment

Client applications today are mostly written as single-user applications. The traditional approach is to re-write it as a client for collaborative application. This can be a formidable task, especially when all possible preferences for modes of collaboration cannot be anticipated in advance. Such a client may turn out to be user-unfriendly or confusing for a particular team. Simple preferences, such as whether and when should the scrollbars track for all participating collaborators, or should separate scrollbars be provided (and color-coded) for each participant, are at the core of such issues [58, 92, 90].

Most collaborative systems follow one of the two widely known architectures: a centralized architecture or a distributed architecture. We will briefly discuss both of these architectures as used in two different collaborative environments.

Virtual network computing (VNC) [54, 55] uses a centralized architecture. It is, in essence, a remote display system which allows one to view a computing ‘desktop’ environment not only on the machine where it is running, but from anywhere on the Internet and from a wide variety of machine architectures. It uses a simple protocol, whereby a server scans the pixel of the displayed image in the frame buffer and asks the remote client to display the same pixel on its screen. Also the pixels are transmitted in a highly compressed format and the updates are sent only when the client explicitly requests for an update. This results in a highly efficient implementation which works reasonably fast even over a slow connection. Collaboration is achieved because the server allows multiple users to connect to the same display screen.

In contrast, GroupKit [92] is mostly a replicated run-time infrastructure. It actively manages the creation, location, interconnection, and teardown of distributed process; communication setup, such as socket handling and multi-casting; and groupware specific features such as providing the infrastructure for session management and persistent conferences. GroupKit is built as an extension to Tcl/Tk that allows the development of groupware applications such as drawing tools, editors, and meeting tools for sharing among several users simultaneously. Figure 4.6 shows the two architectures used in VNC and GroupKit.
We propose a novel client/server architecture for tk-based applications: rendering any single-user application collaborative, without a code re-write. Participants themselves are allowed to dynamically re-configure the inter-client synchronization table to suit their changing preferences and needs. The CollabWiseTk toolkit, based on the proposed architecture, is an extension of the tk functionality to support collaboration. Let us first examine the various issues of rendering a simple text widget collaborative. Consider a very simple application which consists of a text widget and a vertical scrollbar. Such an application allows the user to type in text and the vertical scrollbar allows the user to browse the text information, when the size of the text widget is not large enough to display the entire text at the same time. This application can be very easily built using four lines of tcl code, as shown in Figure 4.7(a), and is a basic
widget used by many complex applications that need functionalities such as syntax highlighted message display, text editing, and html display, to name a few.

Several possibilities exist even for a simple text widget that is rendered collaborative. Figures 4.7(b) - (e) shows various possible collaborative configurations of a text widget for two users Alice and Bob.

1. In Figure 4.7(b), Alice and Bob share the same view of the text widget as well as the scrollbar. A centralized server ensures that both the views are synchronized at all the times. Possibilities of conflict arise when Alice and Bob both try to interact with the text widget at the same time. Such conflicts are typically resolved by some form of locking mechanisms, such as round-robin, first-come-first-serve, user-controlled token passing, etc, so that only one person can interact with the application at a time while the others are forced to watch.

2. Figure 4.7(c) shows a distributed implementation of a collaborative text widget that allows Alice and Bob to interact with their individual text widgets, while an event synchronizer dispatches these interactions to the other users. This mechanism also allows participating users to have different views of the same widget and occasionally ‘glance’ at the other widgets.

3. Figure 4.7(d) shows a configuration where both Alice and Bob get the same view of the text widget, but each has a personal edit cursor so that both can type in simultaneously without affecting the other. In the example shown, Bob has an edit cursor at the top, while Alice has an edit cursor at the bottom and hence both can work on different sections of the same text file. However, the associated scrollbar needs to be configured such that when Bob changes the scroll-view, Alice may not want to follow Bob’s scrolled movement when she’s editing, but may want to do so when merely watching Bob’s interactions.

4. Figure 4.7(e) shows yet another configuration where the text widget is replicated once for every participating user. Therefore, Alice and Bob both have two text widgets - one where each can type in text and another where each can observe
(a) Simple text widget with a vertical scrollbar and its tcl code.

```tcl
    text .txt -yscrollcommand ".sv set" \
        -height 15 -width 50
    scrollbar .sv -command ".txt yview" \
        -orient vertical
    pack .txt -fill both -side left -expand 1
    pack .sv -fill y -side left
```

(b) Synchronized collaborative view using centralized server.

(c) Collaborative view using distributed architecture.

(d) Collaborative shared editor.

(e) Collaborative chat box.

Figure 4.7: Desirable collaborative configurations for a scrollable text widget.
what the other is typing. Here again, Bob may prefer the scrollbars to be synchronized with Alice, while Alice may want to scroll independently Bob’s text widget. This is equivalent to widely available chat tools or the Unix ‘talk’ utility.

All of the above examples can be easily implemented in tcl by re-writing the four lines of single-user tcl code. However, when this text widget is part of a more complex application containing several other widgets, each of which can themselves have their own numerous configuration possibilities, it becomes very difficult to anticipate in advance a suitable configuration preference. Such a collaborative application might turn out to be user-unfriendly or confusing for a particular team.

### 4.3.1 Architecture for Synchronous Collaboration

The CollabWiseTk toolkit consists of two parts: (1) a synchronizing group server (SGS) that provides mechanisms for communication and synchronization among multiple-user client applications; and (2) a distributed synchronizing group client that provides mechanisms for inter-client synchronization among multiple user client applications for effective collaboration. The general architecture of the toolkit is shown in Figure 4.8.

SGS is a tcl server that accepts socket connections from various collaboration clients. It has three types of repositories: (1) *tcl scripts and packages* - various single-user tcl applications are deposited here and are available to users for collaboration; (2) *inter-client synchronization tables* - different configuration preferences for a tcl application are stored in this tables; and (3) *registered users and access permissions*. For security reasons, the latter maintains a list of registered users and their corresponding access privileges.

The collaboration client is installed on each user’s machine and provides an interface to the user for collaborating with other users. Upon invocation, the collaboration client establishes a socket connection with SGS and prompts the user to identify herself. Once the login process is completed, the user can access several different tcl
Figure 4.8: A high level view of client/server architecture for collaboration.

applications, based on her access privileges, by invoking an appropriate configuration file from the inter-client synchronization table. Collaboration clients also maintain a local cache of the tcl applications for faster access.

The collaboration client also provides mechanisms to install new tcl scripts and packages in the group server repository. Privileged users can install such scripts and easily create configuration files on the server for others to access.

When two or more clients access the same configuration file of a tcl package, they are immediately set-up for collaboration. User-interactions with the tcl application are then sent to the group server, which in turn relays this information to all participating users.
Inter-Client Synchronization

Inter-client synchronizer allows the user to dynamically re-configure the different modes of collaboration for every primitive object contained within the tcl application GUI. A primitive object is an element of GUI with which a user can either interact or observe: (1) all tk widgets, such as button, label, entry, etc, are primitive objects; and (2) in addition, tagged items of a canvas and text are also considered as primitive objects.

Primitive objects can be configured into either one of the two states - interact or observe. When a primary object is configured to be in observe state, a user is prevented from interacting with it. For example, a user can be prevented from interacting with a text widget by removing all of its binding tags, as follows:

    bindtags .txt {""}

The same text widget can be brought back to an interact state merely by restoring its binding tags.

When several users work collaboratively, each user invokes the tcl application locally. Therefore, all primitive objects of an application are replicated on each user machine. In addition, a user can now configure her primitive object to be in one of the two states, interact and observe, and link it to any of the participating users. This means that when there are two users, say Alice and Bob, each can configure their text widget to be in one of the four states:

1. Interact/Alice,
2. Observe/Alice,
3. Interact/Bob, and
4. Observe/Bob.

We next explain the meaning of each of these states in the context of Alice's screen:
1. **Interact/Alice**: Alice is configured to interact with her own text widget. At the same time, Bob can observe her interactions, if he has configured his text widget to be in *Observe/Alice* state.

2. **Observe/Alice** Alice is configured to observe her own text widget. This would result in activities on the text widget, unless Bob has configured himself to be in *Interact/Alice* state.

3. **Interact/Bob** Alice is configured to interact with Bob’s text widget. However, if Bob is also configured to be *Interact/Bob* state, then this could lead to potential conflicts.

4. **Observe/Bob** Alice is configured to observe Bob’s text widget.

Similarly, Bob’s screen is in one of the four states and the tk application behaves according to the configuration.

### 4.3.2 Implementation of SGS/SGC

The client-server architecture of the **CollabWiseTk** toolkit is implemented using socket programming. Typically, several clients may be connected to SGS at any time. Clients may invoke several tcl applications and each tcl application may have different collaborative participants. Figure 4.9 depicts one such scenario, where user1 on client1 has invoked TkAppln1, TkAppln3 and TkAppln4, user2 on client2 has invoked TkAppln2 and TkAppln4 and user3 on client3 has invoked TkAppln1 and TkAppln2. An application that is common among users imply that those users are its collaborative participants. For example, TkAppln1 is rendered collaborative among user1 and user3. This is also shown as a dashed line linking TkAppln1 to client1 and client3 on the server side.

The following table provides such a relationship for the example shown in Figure 4.9:
As the number of participants and the number of applications increase, this could lead to very complicated relationships. Also, it is important that the server as well as the client maintains this information in a clean fashion without mixing up any information with one another. Therefore, the synchronous group server invokes a new safe interpreter, using the command `interp create -safe`, for every new client that has requested socket connection and so also for every application that is invoked. Similarly, the client also invokes a new interpreter for every application that the user
invokes.

The SGS/SGC client-server also communicate with one another using a well-defined set of APIs, similar to those used by AGS/AGC. Again, there are basically two types of API commands: (1) synchronous API commands, which block the sequence of execution by waiting for the results; and (2) asynchronous API commands, which do not block and wait for the results of the execution. When client1 sends a user interaction to a collaborative participant client2, it is not necessary for the client1 to verify whether client2 has managed to receive this information or not. Hence, this falls into asynchronous type of command. On the other hand, when client1 requests specific information about client2, such as the status of its scrollbar, then it is necessary to issue a synchronous command. The implementation of SGS/SGC therefore uses the same two procedures, sendNforget and sendNReceive, for communication.

The notification of user interactions with a widget is achieved by analyzing the entire set of commands in tk widgets and re-defining these commands appropriately. Specifically, the original text widget is re-named and a new procedure is created by the same name. This new procedure performs actions necessary: (1) to inform the group server, whenever a new text widget is created and if some other client is observing this text widget; (2) create a new procedure for the text widget pathname that selectively sends similar information to group server.

The following tcl code snippet shows an example of how a text widget is made collaborative.

```tcl
# Rename text widget
#
rename text text-0rg
proc text {pathname args} {
    sendNforget $::sid "text $pathname $args"
    set ret [uplevel 1 text-0rg $pathname $args]
#rename $pathname cmd which just got created
rename $pathname $pathname-0rg
proc $pathname {args} {
    # process $args
    switch -- $opt {
        cget {
```
# send nothing to remote server, OR
# use "sendNreceive $pathname sget"
# cmd to get option value from rmt
# user, when configured for
# interaction with rmt user.
}
insert {
  sendNforget $:.sid \n      "$pathname insert char"
}
};# End of switch stmt
# ....
};# End of proc $pathname
return $ret
};# End of proc text

4.3.3 Application Examples

The simple text widget example with a vertical scrollbar, described earlier and also shown in Figure 4.7(a), can be easily configured into various collaboration modes by defining appropriate object state for the two widgets. Figure 4.10(a) shows a snapshot of a text widget GUI and Figure 4.10(b) shows its corresponding widget tree. This application is readily transformed, without any re-write, into: (1) collaborative shared editor for source codes (Figure 4.10(c), and (2) a chat box window (Figure 4.10(d), by providing an appropriate configuration file inter-client synchronization.

Figure 4.10(e) shows a snapshot of the inter-client synchronization table. It lists all the configuration files for the text widget in the left column, with their respective widget trees. Collaborative participants, if any, are listed in the adjacent columns. Thus, there are three participants for ‘Tk Widgets’, two participants for ‘Shared Editor’ and ‘Chat Box’, and only one participant for ‘Scrollable Text Box’. The entries listed below each user corresponds to the current state of the respective widget. For example, the text widget ‘.txt’ under ‘Shared Editor’ is listed as Interact/lavana for both the users, ‘lavana@cbl.ncsu.edu’ as well as ‘brglez@cbl.ncsu.edu’. This implies that both the users are sharing the text widget. Each such entry can be changed to a different state merely by clicking on the dropdown menus and selecting the desired
(a) Simple scrollable text widget.

```
Text Widget with a Vertical Scrollbar

- Text Buffer: "empty" command "exit set" -height 7
  -width 50
- scrollbar Bspec -command "Chat view" -orient vertical
  pack scrollbar -fill both -side left -expand 1
  pack scrollbar -fill y -side left

lavana@gemini.chl.ncsu.edu 7/25/98 00:00:01
```

(b) Widget tree structure

```
Widget tree
  |_ .  / .f.txt
  |_ .f  / .sv
  |_ .info  / .info.user
         / .info.time
```

(c) Collaborative shared editor.

```
Shared Editor (Source Code)

```

(d) Collaborative chat box.

```
Chat Box (Talk Window)

pack scrollbar -fill both -side left -expand 1
pack scrollbar -fill y -side left
pack scrollbar -command "Chatview" -orient vertical
pack scrollbar -fill both -side left -expand 1
pack scrollbar -fill y -side left
pack scrollbar -command "Chatview" -orient vertical
pack scrollbar -fill both -side left -expand 1
pack scrollbar -fill y -side left
pack scrollbar -command "Chatview" -orient vertical

lavana@gemini.chl.ncsu.edu 7/25/98 00:00:01
```

(e) Inter-client synchronization table

```
lavana.tk

```

Figure 4.10: Scrollable text widget configured as shared editor and chat box.
state.

We also decided to use the Tk widget demonstrations, distributed with the core Tcl/Tk, as a test-bed for testing the CoollabWiseTk toolkit. We have chosen these demos because they not only cover most of the commands in the tk widget set, but also demonstrate usefulness of the toolkit in rendering these applications collaborative. We conducted this experiment by manually invoking all the demos and verify their operation in several different configuration modes.

![Tk Widget Demonstrations](image)

Tk Widget demos are invoked in synchronous collaborative mode. Specifically, it shows a 15-puzzle game which has been made collaborative for two players as described next. First player can only click on the odd buttons whereas the second player can only click on the even buttons. Two kinds of games can be played with such a configuration: (1) players assists one another in completing the game at the earliest; or (2) one player tries to prevent the other player from completing the game.

![15-Puzzle Demonstration](image)

Figure 4.11: Collaborative Tk widget demos.

Our current implementation of the toolkit allows us to render collaborative many of the listed demos. However, we have not yet implemented: (1) one major tk widget,
namely the canvas widget; and (2) the tagged items on the text/canvas widget in our collaboration toolkit. Figure 4.11 shows the main window of the tk widget demonstrations invoked in collaborative mode. Specifically, it shows a 15-puzzle game which has been made collaborative for two players as described next. First player can only click on the odd buttons whereas the second player can only click on the even buttons. Two kinds of games can be played with such a configuration: (1) players assists one another in completing the game at the earliest; or (2) one player tries to prevent the other player from completing the game.

4.4 Web-based Environment for Tcl Applications

The Tcl-plugin, based on Safe-Tcl, restricts running large tcl/tk applications inside a Web-browser. A few of these restrictions are listed below:

- Auto-load scheme fails, unless the application package is installed on the client host. Another alternative is to merge all the scripts in the application into a single script which can be downloaded as a tclet.

- Applications are restricted to a single window since the command toplevel is not available in Safe-Tk and new windows cannot be created.

- Menu widgets are also disabled in Safe-Tk.

- Tclets do not have access to standard input and standard output.

The Tcl-plugin supports multiple security policies so that the tclets can perform any of the functionality described above. However, this requires every client host to devise and customize their security policies for every application before accessing these as tclets.

It is desirable that the generic Tcl applications be easily translated into tclets and made readily available on the World Wide Web:

- without requiring any major changes in the application code, and

- without requiring any sophisticated security policy to run the tclet.
We have developed the WebWiseTclTk toolkit as an enhancement to the Tcl-plugin that makes use of the home policy only. The home policy is, by default, enabled in the Tcl-plugin and hence applications using WebWiseTclTk do not require the host clients to modify their existing security policies.

The toolkit WebWiseTclTk consists of two parts: (1) WebWiseTcl which is an enhancement for Safe-Tcl and is useful for applications that do not require display, and (2) WebWiseTk which is an enhancement for Safe-Tk for applications requiring display. The toolkit itself consists of several smaller scripts and uses the modified auto_load mechanism designed for WebWiseTclTk.

We next describe the architecture of the two packages WebWiseTcl and WebWiseTk.

### 4.4.1 Architecture for Enhancement to Safe-Tcl

A large application, written in Tcl, typically consists of a short main script and a library of support scripts. Applications start up quickly by invoking the main script. As new features are accessed, the code that implements them is loaded automatically, using the auto_load mechanism available in Tcl.

Figure 4.12(a) shows the general architecture that implements the auto_load mechanism. Special cases of the generalized architecture are shown in Figures 4.12(b), (c) and (d) and described below:

1. Typical client host, downloading a tclet from a Web server, has only the Tcl-plugin installed for its Web-browser. The server site provides not only the tclet scripts but also the WebWiseTclTk toolkit as shown in Figure 4.12(b). The client host downloads the main script for the tclet which requests to use the home policy. If the client host has not disabled the home policy, then the main script downloads the initialization script of the WebWiseTclTk toolkit from the server site. Once the initialization has completed, the auto_load mechanism is modified to dynamically download the remaining scripts of the application as and when needed during execution of the tclet.
2. In the second case, shown in Figure 4.12(c), the client host has locally installed the WebWiseTclTk toolkit. The main script of the downloaded tclet uses the

Typical scenario of a tclet execution:
1. Client host visits the Web server.
2. Main tclet script is downloaded to client host.
3. WebWiseTclTk toolkit is loaded and initialized.
4. Main tclet script executes.
5. Individual tclet scripts are downloaded as and when required.
locally available toolkit and visits the server site only to retrieve its other scripts. Thus, this results in faster execution of the *tclet* code.

3. In the third case, shown in Figure 4.12(d), the WebWiseTclTk toolkit is neither available on the server site nor is it installed on the client host. It is available at the software repository site at CBL. This requires the client host to install a special *WebWiseTclTk policy* that allows the *tclets* to download scripts not only from its server site but to also download the toolkit from the CBL site. This mechanism has the advantage of always using the latest version of the WebWiseTclTk toolkit.

The generalized architecture allows the main *tclet* script to dynamically use one of the above three mechanisms, based on the configuration of the client host. We next describe how generic Tcl commands can be modified to overcome these limitations. However, to maintain security, it is important that the unsafe Tcl commands be hidden or restricted in Safe-Tcl. Several different security policies offered by Tcl-plugin 2.0 are convenient and allow the application programmer to design *tclet* codes accordingly. We intend to make use of the *home policy* to enhance the functionality of the Safe-Tcl for WebWiseTcl.

Script libraries and packages provide an excellent mechanism to structure an application code into several smaller scripts, and then dynamically load each script as needed. We modify the restricted commands in Safe-Tcl such that it supports the packaging facility to automatically load scripts from the server site of the *tclet* code. We only need to append the location of the server site, given by "getattr originHomeDirURL", to the *auto_path* variable for the *auto_load* procedure to work correctly with the modified commands described next.

**Source.** The filename argument for the *source* command is parsed for a URL. If the filename is a URL, then it is downloaded using the command "::browser::getUrl filename" and its contents are evaluated. Otherwise, the original *source* command is invoked, as shown in Figure 4.13.

**Open and close.** When a filename specified for *open* is a URL, the specified URL is downloaded and saved on the temporary disk space of the host system assigned by
# Move original command to WebWiseTcl namespace
rename source ::WebWiseTcl::source-Org

# Define a new command in the global namespace
proc ::source filename {
    if {[string match http:* $filename]} {
        # Evaluate script downloaded from a URL
        uplevel 1 [::browser::getURL $filename]
    } else {
        # Invoke original source command
        uplevel 1 ::WebWiseTcl::source-Org $filename
    }
}

Figure 4.13: New definition for command source.

the *home policy*. Then, this file on the local disk is opened and its channel identifier returned. Correspondingly, when a *close* command is invoked for a URL, the file on the local disk is not only closed, but also deleted. These functions are useful for opening a file/URL in read-only mode.

**File.** We have re-defined the command *file* so that its options *dirname*, *join*, and *split* return correct results even when the specified filename is a URL.

**Pwd, cd and glob.** These commands are not available in Safe-Tcl. We therefore assign the URL of the server site, given by "getattr originHomeDirURL", to be the default working directory returned by the command *pwd*. This value is stored in a variable defined in *WebWiseTcl* namespace. The invocation of the command *cd* then results in change of value of the current working directory stored in the variable. The command *glob* returns a list of all matching URLs found under the URL given by the current working directory.

### 4.4.2 Implementation of WebWiseTclTk Toolkit

A complex environment such as the OmniFlow environment, requires that a number of windows be created during its runtime. However, several Tk commands are hidden in
Safe-Tk to prevent denial of service attacks against the host system. This, therefore, limits the scope of the Tcl-plugin to very simple applications consisting of a single window only.

We propose to overcome these limitations as follows:

- re-introduce several of the hidden commands in Safe-Tk,
- use existing commands that are already available in Safe-Tk, to define re-introduced commands,
- change the implications of a few commands, such as "grab -local" and "grab -global".

The following sub-sections describe the methodology used for implementation of the WebWiseTk toolkit.

<table>
<thead>
<tr>
<th>WebWiseTclTk</th>
<th>Preferences</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Area for displaying window icons</td>
</tr>
<tr>
<td></td>
<td>(Text Widget)</td>
</tr>
<tr>
<td>Application Workspace consisting of several windows (Canvas widget)</td>
<td></td>
</tr>
<tr>
<td>On-line help window</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4.14: Layout window of WebWiseTk toolkit.

**Layout.** Figure 4.14 shows the layout window of the WebWiseTk toolkit. It consists of two main widget frames:

- A canvas widget is used to display several toplevel windows that may be created during the execution of a tclet. If the toplevel window is larger than the visible...
canvas area, then scrollbars may be used to display its hidden area. The scroll region of the canvas is initialized to \(1000 \times 1000\) pixels, but may be resized using the preferences option on the top right corner.

- A text widget is used to display button icons for all windows that have been created, including those windows that have been iconified.

Several other widgets are shown, such as the On-line help window, scrollbars for the canvas and text widgets, and preferences to configure the size of the canvas widget. **Toplevel.** The ability to create a detached window, as provided by the command `toplevel`, is very useful for GUI applications of even moderate complexity. We define a procedure called `toplevel` which makes use of the command `frame` to create a detached window and display it on the canvas widget.

For every toplevel window, a set of several frames is created, as shown in Figure 4.15.

![Diagram of window structure](image)

*Figure 4.15: Implementation of a toplevel window using frames.*

This gives the look and feel of a real window that would have, otherwise, been
created by the window manager of the local host system. The frames are laid out using the grid geometry manager. Each frame serves a special purpose:

- The frames on the border have a default color and an active color which is highlighted whenever the mouse cursor moves inside a window. This helps the user to identify the window that is currently active. These frames are also useful for changing the size of the window.

- The next set of frames, just below the resizing frame on the top, provide several functions related to the window, such as kill, iconify, maximize/restore size or display the title of the window.

- A main frame is created in the center corresponding to each toplevel window. All subsequent child windows of the toplevel are packed into this frame.

- If the toplevel window has a Menu Bar associated with it, then the menu items are packed into a frame just above the main frame.

Having created these sets of frames, they are then packed onto the canvas widget in the application workspace area in Figure 4.14 by creating a canvas item of type window. This results in a restriction that the pathname of the window must either be a child of the canvas widget or a child of some ancestor of the canvas widget. Hence, the window names of every toplevel command is translated to a new window name that is a child of the canvas widget. For example, a new toplevel window called ".w" is translated to a new window name called ".c.1.w", where ".c" is the name of the canvas widget and ".c.1" is the name of a unique frame created for encapsulation of the new toplevel window. In addition, we create several bindings to manage and interact with the command wm, described next. A <Destroy> binding is also associated with every toplevel frame so that the entire set of frames is destroyed whenever the toplevel window is destroyed.

**Wm.** The window manager command wm needs to be defined as a procedure which manages the various attributes of the window created using the procedure toplevel described earlier. It can be used to change the title of the window, to iconify/de-iconify the window, or to return the state of the window.
Grab. An indefinite global grab performed by a *tclet* will result in a *denial of service* attack since all the input from the terminal would be re-directed to the *tclet* forever. But, if we re-define the implication of a global grab such that it affects only the windows created by the *tclet*, then it can be considered to be safe.

Thus, the command "*grab -local $win*", as defined here, results in grabbing of a single window within the *tclet* code and the command "*grab -global $win*" results in a grab across all the windows within the *tclet*. This effect of grab can be implemented by associating a new class of bind called *WebWiseTclTk* with every window in the *tclet*, as follows:

```
bindtags $w [linsert [bindtags $w] 0 WebWiseTclTk]
```

Initially, the class *WebWiseTclTk* has no bind scripts associated with any of the events. Whenever a grab is performed on a window, a bind script is created for each event sequence that redirects the event to the grabbed window. The *event generate* command is used to process the event in the grabbed window. Figure 4.16 shows a script that achieves a global grab for a specific window. A grab is released by re-

```bash
foreach seq $AllEventSequences {
    bind WebWiseTclTk $seq "
    # Redirect events to grab window
    if {!\![string match $win* %W] && \n        [winfo exists $win]} {
        event generate $win $seq
        break
    };
 treason if stmt
    "
}; # End of foreach loop
```

Figure 4.16: Script to achieve a global grab on a window.

initializing the bind scripts for the class *WebWiseTclTk* to null. Variables are used to store the state of the grab command and return appropriate values for queries such as "*grab current*" and "*grab status*".
Menus. Menu widgets are as important as any toplevel widgets in any GUI applications, since they allow the user to invoke a list of one-line entries as and when required. The structural layout of the menu widgets created using frame and other Tk commands is shown in Figure 4.17. The command menu creates a toplevel frame and different types of widgets are added inside this frame: button widgets for command entries, checkbutton widgets for check button entries, radiobutton widgets for radio button entries and menubutton widgets for cascaded menu entries. Separator entries are created using frame widgets as shown in Figure 4.17.

![Diagram of menu structure]

Figure 4.17: Implementation of the command menu.

This structure is hidden from the display until the user clicks on the menu button at the top. The implementation of the command grab, as described earlier, is important and allows us to post the menu widget frame whenever the user clicks on the menu button. As the user moves the cursor over different widgets in the menu frame, each widget is highlighted and the associated command invoked if necessary. Clicking on the cascaded entry results in the posting of another menu frame with its associated entries.

If the menu widget is of the type pulldown menu in a Menubar, then the menu entries are packed into the Menubar frame that was created in the toplevel procedure.
Standard I/O and audio. We have created a special window for standard I/O in WebWiseTclTk. Any communication to the standard I/O channel by commands such as puts and gets is redirected to the special window, as shown in Figure 4.18. Therefore, it is possible for the tclet and a user to interact through the commands puts and gets.

Audio commands, such as bell, are still potentially dangerous, with the risk of producing a continuous tone. Therefore, we defined a procedure bell which produces a visual effect by momentarily changing the background color of the canvas widget.

![Diagram of Standard I/O of WebWiseTclTk](image)

Figure 4.18: Standard I/O of WebWiseTclTk toolkit.

Safe commands. Earlier, we noticed that whenever a toplevel window, say ".w", is created, the window name is mapped to a new window name ".c.1.w", corresponding to the main frame in the set of toplevel frames. Therefore, existing safe commands such as button with window names ".w.b" will fail, unless their window names are also translated to a new name ".c.1.w.b", which is in the hierarchy of the toplevel main frame’s children.

We rename the existing safe commands by moving them into a namespace for WebWiseTk, and define new procedures for them. Figure 4.19 shows a sample code for re-defining the command button. The newly defined procedure does the following:

1. maps all the window names, in the arguments passed to the procedure, to the corresponding hierarchy in the toplevel frame.
2. evaluates the original command \texttt{button} with mapped arguments. This creates a new command "$\texttt{new}_w" for the translated window pathname.

3. defines a new procedure for the original window pathname "$\texttt{w}" that would have been created otherwise. This procedure, in turn, invokes "$\texttt{new}_w" whenever it is called.

4. translates the window names in the returned values back to original window names. This is important because the returned values may be directly passed to other code for evaluation. Example: "pack [\texttt{button} .\texttt{b}]".

5. returns the translated value "$\texttt{new}_\texttt{ret}".

```
# Move original command to WebWiseTk namespace
rename button ::WebWiseTk::button-Org
# Define a new command in the global namespace
proc ::button {w args} {
# Map window names
  set new_w [mapArgWindowNames $w]
  set new_args [mapArgWindowNames $args]
  set ret [uplevel 1 ::WebWiseTk::button-Org \n    $new_w $new_args]
# Create a proc called $w
  proc ::$w {args} {
    # Script to invoke command $new_w ...
  }
# Map back returned window names
  set new_ret [mapRetWindowNames $ret]
  return $new_ret
}
```

Figure 4.19: New definition for command \texttt{button}.

The command \texttt{bind} also has to be re-defined. This is because the value of "%\texttt{w}" in the bind script gets the real window name (".c.1.b") instead of the window name
(".b") supplied by the tclet. Thus all window names referred by "%w" in the bind
script are mapped back appropriately, before invoking the original bind script.

Similarly, the command winfo is also redefined, so that its queries, such as "winfo
width", "winfo children", etc., are correctly handled.

Unsafe commands. Few commands, such as send, tk_getOpenFile, tk_getSaveFile,
etc., do not pose the denial of service attacks, but are still unsafe and very dangerous
to the client host system since they present other forms of security attacks. These
commands are therefore not available in WebWiseTk. However, it is always possible
to use an appropriate security policy, other than the home policy, to enable these
commands.

Unsafe options. A few options for safe commands are considered unsafe and hence
not available in Safe-Tk. These include "-bitmap @filename", "-file filename"
and "-maskfile filename", among others. It is possible to allow these options on
the following conditions:

1. the host system supports the use of the home policy, and

2. the specified file exists on the server site of the tclet code.

In such a case, the data for the specified filename is downloaded from the server site
using the command "::browser::getURL filename". Then "-file filename" or
"-maskfile filename" option is replaced by "-data $downloadedData" or "-maskdata
$downloadedData". On the other hand, for the option "-bitmap @filename", a
bitmap image is first created using the command image. Here the replaced option is
"-image [image create bitmap -data $downloadedData]". The command "image
create image -file filename" is also replaced with "-data $downloadedData"
option, after we download the data for the specified filename from the server site.
However, the option "-data $downloadedData" expects the image data to be in
base64 format. Images in other formats are therefore translated to base64 format
using the tcl-only encoding procedures available in the Data Handling Package [116].
The encoding process is slow and hence for images of considerable size, one should
save the original images in base64 format, instead of encoding them on the fly during
execution of the tclet.
4.4.3 Application Examples

In this section, we describe two out of several applications that were readily made accessible within the Web browser using the WebWiseTclTk toolkit. The first example is a general purpose tcl-based WYSIWYG presentation and layout tool, called Impress [117], that can be used to create presentations and postscript documents using fully scalable graphics similar to programs like Macromedia Freehand, Corel Draw, Adobe Illustrator and Visio. This application can be used either as a stand-alone application as within the Web-browser with the tcl-plugin. Although this application comes as a single package, it internally contains two sets of code for each functional operation and it executes the appropriate set of functionality depending on whether the application is run in stand-alone mode or within the Web-browser. The code for the Web-browser is written such that it does not use any of the tk commands, such as toplevel, menus, etc in the application.

We ported the stand-alone mode of Impress application to the Web-browser very easily by writing a small wrapper script. Figure 4.20 shows a screenshot of the Impress application made accessible within the Web browser using WebWiseTclTk toolkit.

The second application consists of porting the OmniFlowLite environment within the Web-browser. Figure 4.21 shows a screenshot of the collaborative document authoring project, described earlier in asynchronous collaboration section, made accessible within the Web browser using WebWiseTclTk toolkit. This project has been successfully used in the senior project class CSC591b and has also been published in [111].

Finally, we also evaluate and describe in detail use of the WebWiseTclTk toolkit in porting to the Web the standard Tk widget demo set that is distributed with Tcl/Tk software.

4.5 Evaluation of Toolkits

This section presents an evaluation of the two toolkits described in this chapter. CollabWiseTk toolkit is compared with two other collaborative systems: VNC and
GroupKit, using several collaboration parameters. To evaluate WebWiseTclTtk toolkit we make use of the entire set of Tk widget demos that are distributed with Tcl and tabulate the results of how demonstrations could be made accessible through the Web browser.

4.5.1 CollabWiseTk Toolkit

We use several collaboration parameters to evaluate and compare the CollabWiseTk toolkit with VNC and GroupKit. For our comparison, we consider two factors: (1) level of collaboration support, and (2) amount of programming effort required to develop the collaborative application. We have created a table for three architectures,
as shown in Figure 4.22, that lists these two entries for each parameter of collaboration. The first entry rates the level of collaboration support available for the specified parameter and a higher value indicates better support. The second entry in the table indicates the amount of programming effort required on part of the developer to create the collaborative application.

We next discuss the collaboration parameters in detail and the corresponding rating for each of the three systems.
<table>
<thead>
<tr>
<th>Collaboration Parameter</th>
<th>VNC Support</th>
<th>VNC Effort</th>
<th>GroupKit Support</th>
<th>GroupKit Effort</th>
<th>SGS/SGC Support</th>
<th>SGS/SGC Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>User-configurability</td>
<td>none</td>
<td>-</td>
<td>low</td>
<td>high</td>
<td>best</td>
<td>low</td>
</tr>
<tr>
<td>Group-awareness</td>
<td>low</td>
<td>high</td>
<td>good</td>
<td>low</td>
<td>good</td>
<td>low</td>
</tr>
<tr>
<td>Floor-control</td>
<td>none</td>
<td>-</td>
<td>good</td>
<td>medium</td>
<td>good</td>
<td>low</td>
</tr>
<tr>
<td>Performance/scalability</td>
<td>average</td>
<td>-</td>
<td>best</td>
<td>medium</td>
<td>average</td>
<td>low</td>
</tr>
<tr>
<td>Synchronization</td>
<td>good</td>
<td>-</td>
<td>average</td>
<td>high</td>
<td>good</td>
<td>-</td>
</tr>
<tr>
<td>Overall</td>
<td>average</td>
<td>low</td>
<td>very good</td>
<td>highest</td>
<td>very good</td>
<td>low</td>
</tr>
</tbody>
</table>

Figure 4.22: Comparison of various systems to support collaboration.

**User configurability**

We define **user configurability** as the amount flexibility that a collaborative system provides, in order to support various collaborative modes that were discussed earlier in this chapter. A good support for user-configurability implies that the user can change the collaboration mode dynamically during runtime.

With VNC, two more users simply share the same view of the application and hence this parameter is not at all supported in it. Therefore, we have not given any rating to the level of programming effort for this parameter of VNC. The GroupKit toolkit basically does not support runtime user-configurability, however, a developer with sufficient programming effort can implement such a flexibility when writing a collaborative application. Accordingly, we have rated low level of support for user-configurability and high level of programming effort.

On the other hand, this is one of the strongest contribution of the CollabWiseTk toolkit. It does not require the programmer to anticipate the needs of the user. Static configuration files for a variety of collaboration modes can be readily created and saved. Additionally, users also have the flexibility to change the mode of collaboration during run-time to suit their specific needs.

**Group awareness**

*Group awareness* in a collaboration environment refers to how well it facilitates co-ordination of the user activities to one another. Provision of vital information
about remote users, such as telepointers to track mouse movement, indication of users joining or leaving a group session results in increased group awareness among remote users.

The level of support for group awareness in the GroupKit environment is very good since it provides telepointers, multi-user scrollbars and information about a user joining or leaving a session. The effort required to provide such group awareness is low. On the other hand, VNC provides a low level of support for group-awareness because although it has highly synchronized displayed activity, it is difficult to associate a specific action with any user because no information is available as to which user performed the action. Moreover, there is no information available as to how many users are currently collaborating in a given session. Therefore, the effort required in increasing group awareness with VNC system is high, because users have to resort traditional approaches such as phone call or e-mail to correspond with one another.

As for the CollabWiseTk toolkit, inter-client synchronization table maintains a list of active sessions corresponding to each application being shared. Additionally, every session not only lists the number of users actively collaborating on an application, but also displays the type of user activity such as interact or observe for a specific widget. This enables other users to determine the status of a particular user in respect to a specific widget and aids in increasing the group awareness significantly. Hence the programming effort required in this case is low.

**Floor control**

Conflicts can arise when two or more users try to interact with the same object simultaneously and therefore a suitable floor control mechanism is necessary to overcome such conflicts by restricting only one user to interact with a object at any time.

VNC is a completely open collaborative system and provides no floor control mechanism. Therefore, we have not rated the effort required in implementing floor control here. GroupKit provides a good level of support for floor-control and the amount of effort required for the developer is medium.
The CollabWiseTk toolkit provides a very fine granularity over floor-control and allows the users to configure the interaction mode down to the widget element, instead of merely allowing control of the entire application. Additionally, users also have the capability to dynamically change and allow other users to interact with a specific widget of choice whenever the need arises. This provides better flexibility in letting users drive the mode of collaboration to suit their needs, rather than a programmer anticipating the collaborative needs to design specific collaborative-aware application and therefore the effort required on part of the developer is minimal.

**Performance and scalability**

The performance and scalability of the system depends on the number of users and how much information is share during a specific session at runtime.

VNC shares all the information with all the users at all times. So, naturally there is a performance hit as the number of users increase. Again, the programming effort required is not applicable since it is not possible to restrict the amount of information that can be shared among remote users. On the hand, GroupKit provides is the most scalable due to its distributed architecture and restricted information sharing, but it does require medium amount of effort to design a scalable application.

The performance of CollabWiseTk toolkit depends on how the users decide to share the information during runtime. If users want to be aware of all the activities of the other remote users, then naturally there will be a performance hit as the number of users increase. However, it is expected that most users will not work in such fully shared mode and hence this system would offer better scalability. Moreover, when users have the flexibility to reduce the number of widgets that are being shared during runtime to improve its performance.

**Synchronization**

The *synchronization* issue arises due when the information among remote users is not available to others immediately and two or more users may modify the shared data simultaneously. This problem does not occur in centralized architectures such
as VNC and CollabWiseTk, however distributed architectures such as GroupKit do require considerable programming effort to overcome synchronization issues.

Summary

Considering the various collaboration parameters outlined above, the VNC system has an average level of collaboration support and requires very little effort in making an application collaborative. However, it does lack some important collaboration features such as user-configurability and floor-control mechanism.

GroupKit, on the other hand, has a very good level of support for collaboration but it does require a considerable amount of effort in making an application collaborative. Moreover, existing single-user applications have to entirely re-written to render it collaborative.

CollabWiseTk also has a very good level of support for collaboration, although it can have a performance hit when the number of users sharing huge amount of information increases. We next evaluate the WebWiseTclTk toolkit.

4.5.2 WebWiseTclTk Toolkit

We decided to use the Tk widget demonstrations, distributed with the core Tcl/Tk, as a test-bench for testing the WebWiseTclTk toolkit. We chose to translate these demos for the World Wide Web because they cover most of the commands of the core Tcl/Tk that are otherwise unavailable in Safe-Tcl/Tk. Figure 4.23 shows the result of posting these demos on the Web and executing them on a host client as a tclet using the Netscape browser. The process of making these demos available on the Web required writing a simple encapsulation tcl-script of about 25 lines.

The TkWidget demo suite consists of several script files. Each script file is used to demonstrate the functionality provided by a set of related commands. The examples in the demo suite range from simple widgets such as labels, buttons and listboxes to more complex widgets including canvases and textboxes. Each demos script consists of several action events associated with it to provide a practical example of how to write commonly used tk scripts.
Figure 4.23: Tcl/Tk widget demos on the Web.

Our experiments consisted of testing the behavior of each of the TkWidget demo script by making it accessible in a Web browser using WebWiseTclTk. Figure 4.24 shows the results of such an evaluation. The first column, under category, lists the
Figure 4.24: Evaluation of WebWiseTclTk with TkWidget demos as testbed.

class of widgets that are being tested, while the second column, under scripts, lists the number of demo script files present in that category. The third column gives the total number of action events, with a break-up of how many events are occurring in each script file. There are a total of 36 demo script files and 104 action events. The last three columns contains the evaluation report of how many action events passed, how many failed and how many were diagonsised correctly when a specific command is not safe. Overall, a total of 91 test cases passed unconditionally, 5 test cases failed to function and reported an error, and 8 test cases failed to function but reported a constructive diagnostic message.

Here’s a summary of test cases that failed to function properly and some comments on it:

1. In Labels suite of demos, it failed to load an image of type ‘ppm’ format, whereas images with ‘gif’ format had no problems. This is because gif images were encoded with base64 format which can then be directly read by the tcl command ‘image’. However, the ‘image’ command does not support similarly formated ppm images.

2. Tk commands such as selection, tk.chooseColor, etc are not implemented in WebWiseTclTk and therefore correctly reports a diagnostic message for these commands.
3. It fails with an error, whenever a tcl command tries to load a bitmap image using "-stipple filename" option. This functionality is not supported.

4. Copy and paste does not work because the selection command is not implemented. It does give a diagnostic message though.

5. All menu items did not work correctly because the grab command is not fully functional yet.

6. The grab command has not been fully implemented to differentiate between local grab and global grab. Hence the demo scripts for grab command did not behave correctly.

There are few other issues, related to performance of WebWiseTelTk, that cannot be easily evaluated with this suite of demo scripts. We next describe these cases and possible solutions.

1. The Tcl-only implementation, mapping window names onto a canvas object, real-time conversion of gif images to base64 format, etc results in degradation of performance. We have tried to minimize these effects by modularizing the code in such a fashion that related tcl procedures are clustered into single script files. Thus, procedures required for implementation of the menu command are loaded only if the command is used by the tclet code.

2. Another area of optimization is possible by improving the efficiency of the procedure to map window names back and forth onto canvas objects, since this is one of the most frequently called procedure.

3. Installing a local copy of the toolkit with the Tcl-plugin on the client host will improve the performance when the distance between the client host and the web-server is large.

4. We also need to improve the reliability of the toolkit by adding sufficient hooks to handle cases when ‘getURL’ is likely to timeout or fail under high network traffic conditions.
4.6 Summary

Three major environments have been developed and implemented: (1) an Omni-FlowLite client/server built on AGC/AGS architecture to support asynchronous collaboration, (2) a CollabWiseTk toolkit built on SGC/SGS architecture to support synchronous collaboration, and (3) a WebWiseTclTk toolkit that enhances the Safe-Tcl environment to readily access tcl application over the Web. Additionally, the architecture for OmniDesk was also described which integrates the above three environments into a single application. However, due to limitation of time, the implementation of the OmniDesk environment is not yet completed.

We have also evaluated the CollabWiseTk toolkit and compared it with the VNC and GroupKit environments. The CollabWiseTk toolkit provides a very good level of support for collaboration, with much less amount of programming effort than the GroupKit architecture although the GroupKit architecture has better performance and scalable. The evaluation of the WebWiseTclTk toolkit consisted of a testbed of 36 demo scripts with a total of 104 action events. The results of evaluation of the toolkit are reasonably good and consisted of 91 pass, 5 fail and 8 diagnostic error messages.
Chapter 5

Taskflow Implementations of Collaborative Projects

With an OmniFlow client, described in Chapter 3, a single user can create a highly interactive and executable program with component programs distributed on the Internet and accessible via a TCP protocol using telnet-, ssh-, http-, or socket-based clients. The program, implemented as a configuration of the OmniFlow client, is represented as a hierarchy of taskflows, programmed by the user. In Chapter 4, we describe an OmniDesk client/server architecture that renders the OmniFlow collaborative in an asynchronous and synchronous mode for two or more users. In this chapter, we examine the potential and the capabilities of the OmniFlow client by implementing, as executable hierarchical taskflows, a number of representative projects that demonstrate the scalability and the versatility of the OmniFlow client.

Specifically, we describe the projects and their OmniFlow implementations in four sections:

OmniFlow Scalability: It is important to understand and anticipate performance limitations of the current implementation in two areas: (1) how much time is required to load the project description and render it in the GUI, (2) how much time is required to schedule and execute all tasks in a given project description, even when time to execute each task has been set to 0. We use the hierarchical project description $EP$, introduced in Chapter 1.3, to construct a number of
taskflow hierarchies ranging from a simple instance of EP to 2, 4, 8 and 16 instances of EP, arranged to execute in parallel and serially. With EP being implemented with 15 tasks with longest path requiring execution of 10 tasks, a serial arrangement of 16 EP instances results in a taskflow containing a total of 2400 tasks and a longest path of 1600 tasks. A parallel construction of a single EP taskflow, 4 serial taskflow constructions, and 4 parallel taskflow construction results in a taskflow of 9150 tasks and a longest path of 1600 tasks. Here, all 9 taskflows are executed concurrently. Results of experiments reveal an exceptionally good asymptotic performance of the task scheduler – lending support for the efficiency of the scheduling algorithm and its implementation.

Distributed OpenDesign Environment: We constructed a distributed VLSI design flow environment, consisting of commercial and university-based tools, some residing at MSU (Mississippi State University), some residing at NCSU. The first (non-hierarchical) version of the environment was based on the OmniFlowLite and included participants from MIT, MSU, and NCSU. In the final version of OmniFlow, a class of simple design specifications is synthesized at NCSU, submitted for placement and routing at MSU, and returned for statistical analysis at NCSU. A single hierarchical taskflow automatically handles all steps in this process – but it can also be accessed by the user for interactive execution and analysis.

Distributed OpenExperiment Environment: We constructed a prototype testbed for distributed experimental design and performance evaluation of graph-based algorithms. In principle, a number of users in three major role models will have contributed to the actual implementation of such an environment. An archivist will have created a shared-access depository space on a centralized server. One or more librarians will have created equivalence classes of data sets, moved them to the shared depository for final characterization and review. A number of experimentalists will have designed domain-specific experiments, running algorithms with librarian’s data on hosts of their own, and moved the solutions to the shared depository for final characterization and review. Archivist would
handle all reviews and would move to a public URL on the Web (after consensus has been reached): (1) domain-specific equivalence class data sets, (2) summary of posted results to date, (3) guidelines for others to participate in the experiments.

* Distributed OpenWriter Environment: * While this project could not be completed with the latest version of the OmniFlow, its initiation played an important role in devising the OmniFlow environment as presented in this thesis. We therefore present a section that introduces the project goals and its status. Major goals of OpenWriter include maintaining a collaborative project directory for a book, an article, or a technical report that are to be compiled in LaTeX on a local hosts by individual contributors and synchronized on the collaboration server by the project editor and all participating writers. Utilities in this environment include support for hierarchical compilation of subsections, sections, and chapters; support for browsing the database on the server for citation keywords and for automated additions of citations to the database; support for file ownership, collaborative editing and compilation of any subsection, section, and chapter, of the project document.

The earlier versions of the OpenExperiment and OpenDesign and environments have been implemented in OmniFlowLite [26] and are described in [28, 111, 112]. The OmniFlowLite implementation has also been demonstrated in the University Booth at the Design Automation Conference in June 2000 and a cross-platform source code executable posted on the Web [113]. The OpenWriter environment has been specified in [114] and the implementation status is presented in the senior project report [115].

5.1 Scalability of Taskflow Rendering and Execution

We use the hierarchical project description $EP$, introduced in Chapter 1.3, to construct a number of taskflow hierarchies.
At the toplevel, OmniFlow EP consists of three single-task instances (DG), (SG) and (PO) and two hierarchical task instances (DR1–DE1) and (DR2–DE2). Both hierarchical task instances, in turn, consist of three single task instances (DR1), (HD1), (DE) and (DR2), (HD2), and (DE) respectively. In addition, the hierarchical task instance (DR2–ER2) is repeatedly invoked three times during execution before invoking (SG). Therefore, execution of OmniFlow EP involves invocation of fifteen single task instances and four hierarchical task instances, after taking into consideration the three repeated invocations of (DR2–ER2). However, the longest path of task executions in EP is actually only ten due to concurrent invocations of the two hierarchical tasks.

In the experiments that follow in this section, we consider that all single task invocations take an equal amount of time for execution. In addition, we also vary the time required for execution of each task for measuring performance of the OmniFlow environment. Execution time for each task is initially set to zero to measure the overall computational overhead of OmniFlow environment. Then the experiment is repeated by doubling the execution times of each task instance, starting with a value of one second to a maximum of sixteen seconds. The entire set of experiments are invoked ten times so as to allow us to measure its statistical performance.

Figure 5.1 (a) tabulates the results of executing the OmniFlow EP ten times, with an execution delay time for each task instance ranging from zero seconds to sixteen seconds. We have also calculated the average, the minimum, the maximum and the standard deviation of execution times required for ten invocations. Since the longest path delay of EP consists of ten task instances, the overall execution time of EP, under ideal conditions, should accordingly be ten times the time required for each single task execution. Therefore, the actual execution times reported in the table consists of the sum of ideal execution time of EP and the additional computational overhead required by the OmniFlow environment.

The second column, with an execution time of ‘0s’ for each task, reports the actual overhead of the OmniFlow environment. With an average value of 4.853 seconds, and a total of fifteen tasks in EP, one can say that each task requires an overhead of 0.32 seconds to process each task. The third column, with an execution time of ‘1s’ for
(a) Execution times of EP sampled over ten invocations.

<table>
<thead>
<tr>
<th>Iter. Num.</th>
<th>Execution time with delay (in seconds) for each task</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0s</td>
</tr>
<tr>
<td>1</td>
<td>4.510</td>
</tr>
<tr>
<td>2</td>
<td>4.642</td>
</tr>
<tr>
<td>3</td>
<td>5.530</td>
</tr>
<tr>
<td>4</td>
<td>5.510</td>
</tr>
<tr>
<td>5</td>
<td>4.996</td>
</tr>
<tr>
<td>6</td>
<td>4.666</td>
</tr>
<tr>
<td>7</td>
<td>4.248</td>
</tr>
<tr>
<td>8</td>
<td>4.865</td>
</tr>
<tr>
<td>9</td>
<td>4.236</td>
</tr>
<tr>
<td>10</td>
<td>5.330</td>
</tr>
<tr>
<td>Avg.</td>
<td>4.853</td>
</tr>
<tr>
<td>Min.</td>
<td>4.236</td>
</tr>
<tr>
<td>Max.</td>
<td>5.530</td>
</tr>
<tr>
<td>Std. dev.</td>
<td>0.481</td>
</tr>
</tbody>
</table>

(b) Execution times of sequential and concurrent EP invocations.

<table>
<thead>
<tr>
<th>OmniFlow</th>
<th>Total Task Instances</th>
<th>LP Task Instances</th>
<th>Overhead Task Instances</th>
<th>Execution time (in seconds) for each task with overhead subtracted from each value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>0s</td>
</tr>
<tr>
<td>EP-1</td>
<td>150</td>
<td>100</td>
<td>51.0</td>
<td>90.2</td>
</tr>
<tr>
<td>EP-s2</td>
<td>300</td>
<td>200</td>
<td>123.6</td>
<td>187.3</td>
</tr>
<tr>
<td>EP-s4</td>
<td>600</td>
<td>400</td>
<td>255.3</td>
<td>351.2</td>
</tr>
<tr>
<td>EP-s8</td>
<td>1200</td>
<td>800</td>
<td>437.5</td>
<td>778.3</td>
</tr>
<tr>
<td>EP-s16</td>
<td>2400</td>
<td>1600</td>
<td>922.8</td>
<td>1438.2</td>
</tr>
<tr>
<td>EP-c2</td>
<td>300</td>
<td>100</td>
<td>118.8</td>
<td>80.1</td>
</tr>
<tr>
<td>EP-c4</td>
<td>600</td>
<td>100</td>
<td>232.9</td>
<td>70.9</td>
</tr>
<tr>
<td>EP-c8</td>
<td>1200</td>
<td>100</td>
<td>491.9</td>
<td>71.4</td>
</tr>
<tr>
<td>EP-c16</td>
<td>2400</td>
<td>100</td>
<td>979.5</td>
<td>49.3</td>
</tr>
<tr>
<td>EP-all</td>
<td>9150</td>
<td>1600</td>
<td>3484.0</td>
<td>1242.6</td>
</tr>
</tbody>
</table>

(c) Representative times for XML parsing and GUI rendering.

<table>
<thead>
<tr>
<th>OmniFlow (file size)</th>
<th>XML reader + parser task defs. time</th>
<th>GUI loader + render single time</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSU-mentor (13Kb)</td>
<td>5 1.7s</td>
<td>6 1 1.9s</td>
</tr>
<tr>
<td>Parabola-file (16Kb)</td>
<td>6 1.9s</td>
<td>10 1 2.7s</td>
</tr>
<tr>
<td>EP-all (22Kb)</td>
<td>19 2.0s</td>
<td>2 10 3.2s</td>
</tr>
<tr>
<td>DoE-3-way (44Kb)</td>
<td>14 1.8s</td>
<td>9 5 4.1s</td>
</tr>
</tbody>
</table>

Figure 5.1: Results of conducting a series of taskflow experiments.
each task, reports the sum of an ideal execution time of ten seconds of EP and the additional overhead of the OmniFlow environment. Thus, with an average value of 13.938 seconds, if we subtract the ten seconds of ideal execution time, we get the computational overhead of 3.938 seconds. This is comparable to the value reported in the second column. Moreover, all the remaining columns in the table report a similar value.

Therefore, we can safely deduce from these experiments that the computational overhead required for processing each task remains constant and is independent of the actual execution time required for each task.

We next perform a series of taskflow experiments in various configurations so as to determine the scalability of the OmniFlow environment. We have constructed a number of taskflow hierarchies, as follows:

EP-1: This taskflow is the same as EP, except for the difference that this task is repeatedly invoked ten times during the experiment instead of only once for EP. Accordingly, this results in a total of $15 \times 10 = 150$ task instances with the longest path delay of $10 \times 10 = 100$ tasks.

EP-s2: This taskflow is created by connecting two task instances of EP in series and then repeating the sequence ten times during the experiment. This results in a total of $15 \times 2 \times 10 = 300$ task instances and the longest path delay of $10 \times 2 \times 10 = 200$ tasks.

EP-s4: This taskflow is created by connecting four task instances of EP in series and then repeating the sequence ten times during the experiment. This results in a total of $15 \times 4 \times 10 = 600$ task instances and the longest path delay of $10 \times 4 \times 10 = 400$ tasks.

EP-s8: This taskflow is created by connecting eight task instances of EP in series and then repeating the sequence ten times during the experiment. This results in a total of $15 \times 8 \times 10 = 1200$ task instances and the longest path delay of $10 \times 8 \times 10 = 800$ tasks.
EP-s16: This taskflow is created by connecting sixteen task instances of EP in series and then repeating the sequence ten times during the experiment. This results in a total of $15 \times 16 \times 10 = 2400$ task instances and the longest path delay of $10 \times 16 \times 10 = 1600$ tasks.

EP-c2: This taskflow is created by connecting two task instances of EP in parallel and then repeating both the tasks ten times during the experiment. This results in a total of $15 \times 2 \times 10 = 300$ task instances but the longest path delay still remains the same as $10 \times 10 = 100$ tasks.

EP-c4: This taskflow is created by connecting four task instances of EP in parallel and then repeating all the tasks ten times during the experiment. This results in a total of $15 \times 4 \times 10 = 600$ task instances and the longest path delay of $10 \times 10 = 100$ tasks.

EP-c8: This taskflow is created by connecting eight task instances of EP in parallel and then repeating all the tasks ten times during the experiment. This results in a total of $15 \times 8 \times 10 = 1200$ task instances and the longest path delay of $10 \times 10 = 100$ tasks.

EP-c16: This taskflow is created by connecting sixteen task instances of EP in parallel and then repeating all the tasks ten times during the experiment. This results in a total of $15 \times 16 \times 10 = 2400$ task instances and the longest path delay of $10 \times 10 = 100$ tasks.

EP-all: This taskflow invokes all the taskflow instances created above, where each task instance is repeated 10 times. However, EP-all itself is invoked only once. Also, the total number of task instances in EP-all is the sum of all the task instances above which is 9150 tasks and the longest path delay is equal to the longest path delay of the above tasks, which is 1600 tasks.

Again, we measure the execution times of each of the above taskflow configuration by varying the execution time required for each task from 0s to 16s. Figure 5.1 (b) tabulates these results. For each taskflow configuration, it shows the total number
of task instances in the the second column, the longest path of task instances in the third column and the computational overhead of the OmniFlow environment in the fourth column. It is evident from the values in the fourth column that as the number of EP task instances increases, the overhead also increases proportionately. Also, in general, a number of concurrent tasks require more overhead than that for the same number of sequential tasks. For example, EP-s16 takes 922.8 seconds whereas EP-c16 takes 979.5 seconds.

The remaining columns, fifth through ninth, shows the time required for actual execution for each, after subtracting the corresponding value of overhead from the fourth column. The execution times simply increase proportionately for the sequential configurations of EP. For example, EP-s2 takes 187.3 seconds to execute when the execution delay of each task is ‘1s’ and it increases to 388.7 seconds when the execution delay of each task ‘2s’. Similarly, the execution time for EP-s4 is 351.2 seconds and 749.8 seconds respectively. These values are approximately equal to number of the longest path task instances multiplied by the execution time for each task instance.

In contrast, the execution times of concurrent EP configurations is not approximately same as the number of longest path task instances. This discrepancy is due to the fact that task instances scheduled for invocation on the same host does not execute concurrently since only one task on given host is executed at any given instance. This gives rise to a slight increase in the execution times as we move from EP-c2 to EP-c16.

Therefore, in general, results of these experiments reveal an exceptionally good asymptotic performance of the task scheduler – lending support for the efficiency of the scheduling algorithm and its implementation.

Finally, we have also measured the time required for the OmniFlow environment in parsing the XML files containing task definitions and that for rendering the graph in a GUI. Figure 5.1 (c) shows the results of loading the four OmniFlows of varying file size. The MSU-mentor flow consists of five task definitions which takes approximately 1.7 seconds to parse the XML file. Correspondingly, it consists of six single-task instances and one multi-task instance, which takes about 1.9 seconds to render it in a GUI. As the size of the file and the number of task definitions and the number of
task instances increases, the time required for XML parsing and GUI rendering also increase proportionately.

5.2 OpenDesign Taskflow Implementations

The major goal of OpenDesign taskflow environment is to allow users to configure the environment on their own as needed by the project at hand:

- by choosing the best or the most affordable tools for each design tasks – without having to install them on local host;

- by choosing the most effective sequences of tasks to be executed, not only for manual one-task-at-a-time execution but also for scheduling any number of tasks for automated execution;

- by choosing the hierarchy of data structures and revision control most appropriate for the project-at-hand – and having it readily accessible on the Web;

- by creating and storing preferences on the modes of collaboration among the members of a project team.

Rather than implementing a dedicated client/server environment where software developers make most of the choices of how the environment is to be used by designers, we argue in this thesis that the taskflow programming environment such OmniFlow and the collaborative features of OmniDesk will support the creation of user-defined environment such stated above. Here, designers/users rely on a generic client interfaced to a generic server and implement the environment themselves by writing a project-specific configuration file and a set of encapsulation scripts for each tool. As long as each tool also can be invoked remotely with a command-line script rather than the nominal GUI, our demos have shown that a number of project-specific environments can be readily created by re-configuration of the the generic clients such as OmniFlow and OmniDesk.

OmniFlowLite/OmniDesk. OmniFlowLite is first implementation of the user-configurable client, linked to OmniDesk’s Asynchronous Group Server (AGS) to sup-
(a) A task-data graph for a typical system design.

(b) An example of OmniFlowLite/OmniDesk client/server configuration.

Figure 5.2: A task-data graph and its OpenDesign environment implementation.

port file protection and synchronization between collaborating users [26, 27]. We used it to configure an example of an OpenDesign environment as illustrated in Figure 5.2. Here, the environment captures a small multi-team project depicted as a design flow, modeled with a task-data graph.
A task-data graph captures the tasks and data dependencies in a design such as shown in Figure 5.2a. For simplicity, we show an example where chaining all task dependencies creates a single path. The path may be broken into a number of segments; there are two segments in the example. The first segment consists of three simple tasks: optimize netlist, translate netlist, partition netlist. The second segment consists of two iterative tasks: foreach partition translate, foreach partition place & route. Note however, the data dependencies. Some tasks depend on primary input data only, some tasks depend on data generated by the preceding task only, but in general, tasks may depend on data generated by any task preceding the one being executed.

The basic concepts of the OmniFlowLite/Omnidesk architecture are shown in Figure 5.2b: application-transparent universal and asynchronous group server (AGS), tool servers readily accessible from AGS, AGS-based project directories, each created initially by the project coordinator. The project directory (projectDir on the server) has a number of subdirectories that are project-specific and are devised by the team in the course of the project. Only project coordinator has read/write (r/w) permissions to Admin and Resources. Team members can read Resources as well as any Users subdirectory. Each team member has r/w permission in their own subdirectory.

The client received by each team member is enabled once the user enters userID, password, and projectDir. Once enabled, the client configures to a default view that may invoke a pre-configured task flow such as shown in Figure 5.2b. The facility to organize and configure a set of design objectives into a number of task flows is one of the important generic features of this client. For example, Alice’s Flow involves three tasks that are to be executed in the following sequence:

\[
\text{optimize} \quad \text{translate} \quad \text{partition} \\
\text{netlist} \Rightarrow \text{netlist} \Rightarrow \text{netlist}
\]

Clicking on each of the task buttons invokes a tool that may reside on a local host or a remote host. Files produced by one tool may or may not be used as inputs to the tool that is to be executed later. After completion of a task, user may click on the next button in the sequence shown. Alternatively, the interface allows user
to configure on any of the task connectors that can be toggled between $=\Rightarrow$ such that a complete task chain gets invoked on the click of the first button in the chain. This is the state of Alice’s Flow shown in Figure 5.2b; the sunken state of the third button (technology mapping) indicates that the task is being executed.

The files read and written by specific tasks can be filtered for display in the respective file boxes: one box displays the working directory on the remote host, the other on the local host. Notably, if the file is owned by the server (remote host), ownership can be asserted for each file through the interface: either by clicking on the owner (listed as server) or by highlighting the file in the file box and clicking on the lock utility button between the file boxes. Similarly, only file owner can relinquish the ownership to the server, at which point it can be claimed and edited by another team participant.

**Demos at DAC’1999/DAC’2000.** Two releases of OmniFlowLite/OmniDesk environment were demonstrated during the successive Design Automation Conference University Booth Exhibitions in June 1999 and June 2000. They involved participants from MIT and MSU, linking tools and data from MIT, MSU, and NCSU into a number of collaborative design flow configurations, similar to the ones shown in Figure 5.2 [28]. In addition, experimental design flows that were generated as part of a senior design project CSC499 during Spring’2000 [111] were also demonstrated by one of the seniors hosted to attend the conference during June 2000.

**OmniFlow (latest version).** The latest version of OmniFlow (as described in Chapter 3) represents a significant extension of OmniFlowLite. While the OmniFlowLite served well to illustrate the collaborative features of OmniDesk, it only supports an environment where a linear chain of component tools can be combined into a single executable program. There is no support for hierarchical encapsulation of components nor is there a scheduling engine to support arbitrary interconnection of components and corresponding synchronizing elements, i.e. the support not only for any programming construct, including iteration, but also the support for concurrency. However, the number of isolated chains that can be included in a single flow is not restricted.
To demonstrate the capabilities of the latest version of OmniFlow we extended one of the MSU-NCSU design flows captured earlier only in limited capacity in OmniFlowLite. Specifically, we designed an experiment that allows us to submit a number of randomly placed instances from a netlist equivalence class to a specific placement and routing tool at MSU, and after an optimized placement, evaluate variations in parameters such as WLe, TRe, and VIAs. The purpose of documenting the details of this experiment in the thesis is twofold: (1) to demonstrate feasibility of a non-trivial experimental design configuration that fully automates a number of tedious and error-prone tasks with a remote tool that would have to be done manually otherwise, (2) to provide a useful OpenDesign template for distributed design and experiments that may engage other participants later.

As shown in Figure 5.3, the top-level taskflow contains three instances of (mentorflow). In this example, each instance can be considered as a treatment in the sense that we want to evaluate and compare the quality of three placement solutions under different arrangements of input data and parameters that control the quality of cell placement. All edges from the Begin node are disabled in the initial set-up so that each of the instances is invoked with a separate user click, to pursue the following objectives:

(mentorflow-tr00): The main objective is to evaluate the quality of a randomly selected placement. We read the netlist class of identical circuits and the randomly pre-assigned linear placement for cells in each netlist class instance. The tool evaluates this placement ‘as is’, performing no placement optimization.

(mentorflow-tr01): The main objective is to let the tool optimize the placement upon reading a netlist description (no explicit placement data is given). However, we create a netlist class of identical circuits where the order of cells in the netlist description is randomized – the tool may or may not be sensitive to this arrangement. We configure the tool to return an optimized linear placement of cells for each class instance.
This hierarchical taskflow is an upgrade of the MSU-NCSU design flows captured earlier. The three instances of mentorflow are configured to execute three taskflows, each repeated 32-times for different netlist instances from the netlist equivalence class as described in the body of the text. Each mentorflow expands into four components as shown below. For more details, see Figure 5.4.

Figure 5.3: OpenDesign taskflow invoking a web-based component tool at MSU.

(mentorflow-tr02): Again, the main objective is to let the tool optimize the placement upon reading a netlist description (no explicit placement data is given). Here, we create a netlist class of identical circuits where the order and the labels of cells in the netlist description are randomized – the tool may or may not be sensitive to this arrangement. We configure the tool to return an optimized linear placement of cells for each class instance.

These are very basic experiments that should be considered whenever we plan to evaluate a set of performance parameters returned by a new blackbox component tool. Ideally, the tool should return identical placements for both treatments (tr01 and tr02), and on the average, we expect treatment tr00 (random placement) to be
much worse than the placement optimized by the tool. However, since the placement problem is NP-hard, solutions produced by the tool are expected to vary with variation of the netlist input order. If the observed variance of placement is 0, the most likely reason is that the tool itself reorders the netlist internally to a fixed order, regardless or the input order of data – which can happen if the input data is only re-ordered but not relabeled. Hence, in tr02, we also consider the case where both the order of netlist as well as the labels of netlist nodes are randomized for each class instance. Past experience has shown that lack of such precautions can ruin a number of experiments which have to be repeated with new data sets – data that are at this point both re-ordered \textit{and} relabeled [7, 11, 16].

Nominally, the placement tool at MSU is accessible only as part of the web-based interface introduced in Figure 1.3 earlier. However, performing a series of repetitive tasks such as described above with the direct-web interface would not be practical. We created a client that encapsulated the web form, so data all data inputs could be entered as part of the taskflow description. The full context of the encapsulation and the flow implementation is shown in Figure 5.4.

A statistical summary of the experiments with the taskflow in Figure 5.3 is shown in Figure 5.5. The observations that can be drawn from the data as shown are:

- the random placement is on the average not significantly worse than the placement generated by the tool, the minima report approach or even improve on the minima reported by the tool.

- both reordering \textit{and} relabeling of the netlist are required in order to induce the widest range of variations as reported by the tool. Ideally, the tool should report zero variance for all cost function parameters that are being evaluated.

This experiment has been designed with a netlist of only nine cells intentionally – allowing us to focus on the principles of distributed computing rather than the evaluation of the tool itself. One should not draw conclusions that the tool delivers a placement that is not much better than a random placement – the netlists are too small. This experiment is but a template for a additional series of experiment with
The web-form on the left is the nominal interface to the placement tool, supported by the JavaCAD client at MSU [32]. Here, we access it indirectly as an encapsulated web-based component of the taskflow in Figure 5.3, in the context shown below: pargen (local component, generates parameters expected by the web client), mentorWeb (remote component, reads netlist and and invokes a placement tool), postproc (local component, postprocesses the data returned by mentorWeb), extrstats (local component, creates a statistical profile of extracted data).

Figure 5.4: Context for an encapsulated web-based component in a taskflow.

class instances of much larger circuits that is to follow after the completion of the thesis. Not only random but also user-supplied placements will be submitted and compared to the placements generated directly by the tool.
This summary is for the equivalence classes of nineCellAdder

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Sample sizes per row: 33</th>
<th>Population Mean</th>
<th>Sample Mean</th>
<th>Sample Std. Dev.</th>
<th>Sample Min.</th>
<th>Sample Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wle_tr00</td>
<td>36.1664e-01, 5.3579e-01</td>
<td>5.2567e-01</td>
<td>2.8520e-02</td>
<td>3.7000e-01</td>
<td>5.3700e-01</td>
<td></td>
</tr>
<tr>
<td>TR_e_tr00</td>
<td>7.5275e+00, 8.0465e+00</td>
<td>7.3879e+00</td>
<td>7.3983e-01</td>
<td>5.0000e+00</td>
<td>9.0000e+00</td>
<td></td>
</tr>
<tr>
<td>VIA_e_tr00</td>
<td>2.5929e+01, 2.7042e+01</td>
<td>2.6485e+01</td>
<td>1.6835e+00</td>
<td>2.3000e+00</td>
<td>2.9000e+00</td>
<td></td>
</tr>
</tbody>
</table>

Wle ... wirelength  TR00 ... random placement supplied by the user
TR_e ... tracks     TR01 ... tool placement (reordered netlist class)
VIA_e ... vias      TR02 ... tool placement (reordered/relabeled netlist class)

Figure 5.5: Statistical summary of an experiment with the OpenDesign taskflow.

5.3 OpenExperiment Taskflow Implementations

Experimental design is a well-defined discipline in agriculture, biomedical research, and manufacturing [118]. This is not the case when we evaluate and compare the performance of graph-based algorithms. In this section, we introduce and report on an OpenExperiment environment that allows, using the Internet, a dispersed group of peers to collaboratively design, execute, archive, and evaluate the performance of specific algorithms – all in the nominal context of experimental design.

Background and Motivation. The medical school's tutorial [119] in Figure 5.6(a) provides the context for the experimental design task flow in Figure 5.6(b). In order to emphasize the relationship to experimental design, we adopt the standardized terminology of ‘applying treatment’ rather than ‘invoking algorithms’. For algorithms, typical objective of a treatment may be the minimization of a specific cost function associated with an ‘equivalence class’, the latter being the ‘sample subset of the population of graphs’, eligible for treatment. The generation of well-defined equivalence classes of graphs is itself an emerging field, see [4, 5, 6, 8, 10, 14, 120, 121, 122].

The execution of primitive tasks in the flow of Figure 5.6(b) defines the key re-
Figure 5.6: Experimental design flows from two distinctive domains.

requirements for the OpenExperiment environment.

Definition 1 (OpenExperiment primitives).

- On each local host:
  1. download class data
  2. apply treatment
  3. upload solutions

- On the centralized remote server host:
  1. archive graph equivalence classes
  2. evaluate submitted solutions
  3. summarize evaluations

By maintaining a centralized database of all classes and all solutions for each treatment in a given class, the environment supports evaluation of each solution and
updates the evaluation summaries automatically on the Web. The access to the common evaluation routine not only ensures valid comparisons of all treatments but also provides a valuable second opinion on the calibration of cost functions independently optimized by each treatment.

The illustrative histograms summarizing treatments 12 and 17 in Figure 5.6(b) report on the crossing number of a relatively small 2-layer graph [123]: 17 nodes at layer 0, 10 nodes at layer 1, and 50 edges. In the case shown, the class of 64 graphs is formed by randomly permuting the order in which they appear in the file, i.e. the class is an isomorphism class. Without any treatment, the expected crossing number for the total of $10! \times 17!$ arrangements of the 2-layer graph is 522, according to the formula given in [123]. Indeed, evaluating the 64 instances in the class returns a nearly normal distribution of crossing numbers with a sample average of 531.8 and a standard deviation of 45.6, giving rise to 95% confidence interval [118] for the true mean: [520.4, 543.1]. The expected crossing number of 522, predicted by the formula, is clearly in this interval.

The behavior of the two algorithms, denoted as Treatment 12 (implementing the algorithm ‘dot’ as described in [124]) and Treatment 17 (implementing the algorithm ‘GBFS+BC’ as described in [13]) is characteristic of just about any algorithm attempting to solve an NP-hard problem in polynomial time. Particularly for the isomorphism graph class we can readily observe: (1) the performance of two algorithms can differ significantly, (2) most any algorithm is very sensitive to the choice of the input order of the graph. The variance of 0 for treatment 17 is an exception for this particular graph class. Note also, that while the ratio of average of the two treatments is $96.5/66 = 1.46$, the ratio of the worst solution (treatment 12) to the best solution (treatment 17) is $186/66 = 2.82$. This simple example illustrates the merits of applying experimental design methodology to performance evaluation of graph-based algorithms.

**OpenExperiment and OmniFlow.** The goal of OpenExperiment is to simplify the configuration and use of experimental designs. We demonstrate a prototype of an environment using the OmniFlow client and to make experimental scaffolding as
well as the data easily available to any researcher.

Using OmniFlow, we constructed a prototype testbed for distributed experimental design and performance evaluation of graph-based algorithms. In principle, a number of users in three major role models contribute to the actual implementation of such an environment. An archivist creates a shared-access depository space on a centralized server. One or more librarians creates equivalence classes of data sets, moves them to the shared depository for final characterization and review. A number of experimentalists design domain-specific experiments, running algorithms with librarian’s data on hosts of their own, and move the solutions to the shared depository for final characterization and review. Archivist handle all reviews and move to a public URL on the Web (after consensus has been reached): (1) domain-specific equivalence class data sets, (2) summary of posted results to date, (3) guidelines for others to participate in the experiments.

A top-level view of the testbed is shown as a taskflow in Figure 5.7. Notably, the same client taskflow configuration is distributed to all participants: an archivist, a librarian, and three experimenters. However, each participant can only execute, on local and remote host, specific pre-assigned taskflows. To focus the presentation in this thesis, we assume that the domain of the experiment is minimization of the crossing number in two-layer graphs: an NP-hard problem [125]. In other words, the objective is to find permutations of nodes on both layers such that the crossing number is minimum.

Specific roles of three participating groups are as follows:

- The librarian, working on local host (curly@duke), prepares equivalence class of graphs for all participating experimentalists. His taskflow is shown in the mid-part of Figure 5.8. After classes have been generated, the initial (random) placement of each class instance is characterized by the crossing number, evaluated on the remote host (zodiac@cbl) by the crossing number evaluator that is the same for all participants. Once completed, e-mail is sent to the archivist.

- The archivist working on local host (curly@duke), moves all data generated by librarian on the remote host (zodiac@cbl) to the archival site to be accessible
The prototype OpenExperiment taskflow environment brings together a number of participants in three major role models: an archivist who will have created a shared-access depository space on a centralized server; one or more librarians who will have created equivalence classes of data sets, moved them to the shared depository for final characterization and review; and a number of experimentalists who will have designed domain-specific experiments, running algorithms with librarian’s data on hosts of their own, and moved the solutions to the shared depository for final characterization and review. Archivist would handle all reviews and would move to a public URL on the Web (after consensus has been reached): (1) domain-specific equivalence class data sets, (2) summary of posted results to date, (3) guidelines for others to participate in the experiments.

The illustrative taskflow prototype in this figure brings together 5 participants on 4 hosts: an archivist on curly@duke, a librarian on curly@duke, experimentalist1 on gemini@cbl, experimentalist2 on euler@cos, and experimentalist3 on jupiter@cbl. Here, all hosts except jupiter are running under UNIX, jupiter is running under MacOS. The client in this figure would also execute on hosts under LINUX and Windows/NT. All components in this taskflow expand into more detailed tasks, discussed in the body of the text and shown in additional illustrations.

Figure 5.7: OpenExperiment taskflows, bringing together distributed participants.
**Archivist** creates a shared-access depository space on a centralized server and maintains liaison between the participants. With the taskflow such as below, he downloads archives of data sets and evaluated solutions from all participants (librarians and experimentalists) in the experiment, expand them, generates an updated statistical summary of results, and posts all updated archives for public access on the Web.

**Librarian** creates equivalence classes of data sets on the local host and moves all data sets to the shared depository on a remote host for independent characterization and review.

**Experimentalist** designs domain-specific experiments, executes the algorithms with librarian’s data on local host, and moves the solutions to the shared depository on a remote host for independent characterization and review.

Figure 5.8: Principal role models, taskflows for the OpenExperiment environment.

from a public URL on curly@duke. His taskflow is shown in the top-part of Figure 5.8. He also generates the first summary of crossing numbers generated as ‘treatment 0’ by the librarian. Since these are crossing numbers based on random placement, placements reported by algorithms later are expected to be
significantly better – on the average at least. Later on, when experimentalists send mail, archivist will update the statistical summary of results as needed.

- *The experimentalists* working on respective local host (gemini@cbl, euler@eos, jupiter@cbl), dedicate their efforts to executing their respective algorithms, differentiated by a treatment number (Tr12, Tr17, Tr30). An expanded taskflow, e.g. (Expt_Tr17) is shown in the bottom-part of Figure 5.8. Once completed, each experimentalists moves the solution data to the common evaluator on the remote host (zodiac@cbl) and generates an evaluation report (of crossing numbers) for each solution instance. Once the evaluations are completed, e-mail is sent to the archivist.

While the algorithms are in general presented as blackbox implementations, we outline few levels of the hierarchy that make up the treatment Tr30, shown in Figures 5.9 and 5.10. The notable part of this algorithm is that the implementation itself consists of distributed components – with experimentalist developing the core algorithm under MacOS [126], while a relatively stable special purpose permutation component resides and executes on a remote UNIX host [127].

A statistical summary of the experiments with the taskflow in Figure 5.7 is shown in Figure 5.11. The text box in the figure also summarizes the flow of data generation during the different phases of the experiment and the actual data posted on the website. The grand statistical summary illustrate the range of crossing numbers obtained for instances of graphs under different solutions, generated by two algorithms, tr0012 and tr0017, each initialized with the the same random placement reported as tr0000. Observation that can be made from the summarized data include:

- With the average crossing number of 125.2, the random placement is significantly worse than placements generated by the two algorithms, reporting crossing number averages of 21.9 (tr0012) and 30.8 (tr0030).

- For the comparable cost of computation, the algorithm labeled as tr0012 (based on an installed version of a drawing package ‘dot’ [124]) is also significantly
**Experimentalist 3** is responsible for maintaining/executing the expanded component (Expt_Tr30) (introduced in Fig. 5.7). Since he is executing the OmniFlow client under MacOS (jupiter@cbl), and AppleScripts to unzip/zip files automatically are unavailable, instances of OmniFlow library components (DownloadDotOrd) and (uploadTr30Sols) implement the tasks of downloading data and uploading solution instances without data compression. The component instance (evalTr30Class) allows the experimentalist to re-evaluate solutions independently on a remote host (zodiac@cbl). The focus of attention is the implementation of the (MonteCarlo) component, expanded below.

---

**Experimentalist 3** is responsible for maintaining/executing the expanded component (Expt_Tr30) (introduced in Fig. 5.7). Since he is executing the OmniFlow client under MacOS (jupiter@cbl), and AppleScripts to unzip/zip files automatically are unavailable, instances of OmniFlow library components (DownloadDotOrd) and (uploadTr30Sols) implement the tasks of downloading data and uploading solution instances without data compression. The component instance (evalTr30Class) allows the experimentalist to re-evaluate solutions independently on a remote host (zodiac@cbl). The focus of attention is the implementation of the (MonteCarlo) component, expanded below.

... expanding component (MonteCarlo)

... expanding (loopMonteCarloGraph2L)

... expanding (permuteMonteCarloGraph2L)

The Monte Carlo component implements a random permutation search, evaluates a cost function on the 2-layer graph (here, the crossing number) and updates the minimum cost solution. This well known method is a prelude to a more elaborate search algorithm that is currently under construction [126]. A number of user-specified calls are made to (permuteMonteCarloGraph2L) component by (evalMonteCarloGraph2L) before a decision is made to exit with the minimum cost for the netlist instance being optimized (see also Figure 5.10).

Figure 5.9: An expanded view of the taskflow configuration that implements TR30.
The main purpose of the prototype implementation in Figure 5.9 is to demonstrate the capability of developing a new algorithm with distributed components on the network. Here, a special purpose permutation component [127], now also available as a CGI-script on the Web as shown, is invoked as an encapsulated component, delivering a permutation set in a well-defined permutation class or as a set of random permutations.

A number of user-specified calls are made to (getPermutation) component within the taskflow before a decision is made (at an upper level) to exit with the minimum cost for the netlist instance being optimized.

Figure 5.10: A networked permutation component encapsulated in an algorithm flow.

... the CGI-script GUI to execute `permute`

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Your E-mail Address:</td>
<td><a href="mailto:jrgles@bl.monsu.edu">jrgles@bl.monsu.edu</a></td>
</tr>
<tr>
<td>Reference Permutation:</td>
<td>abcdefghpq</td>
</tr>
<tr>
<td>Number of Permutations (p):</td>
<td>500</td>
</tr>
<tr>
<td>Ported or All (d)?</td>
<td>All</td>
</tr>
<tr>
<td>Maximum Inversions (m):</td>
<td>45</td>
</tr>
<tr>
<td>Number of Inversions (-optional):</td>
<td></td>
</tr>
<tr>
<td>Output filename:</td>
<td>untitled perm</td>
</tr>
</tbody>
</table>

As a taskflow instance of `(getPermutation)`.

- While the flow has been set-up to include a contribution from tr0017, we expect that this treatment [13, 15] to be best for the class. If time permits, Dr. Stallmann may execute the given OmniFlow client on his local host and add to the posted results.
The web-page shown on the left is an example of the experimental design directory maintained by the Administrator of the OpenExperiment taskflow such as shown in Figure 5.7. Here we see a total of 33 instances from the netlist equivalence class (files *.dot), generated from the reference circuit representation (file nineCellAdder.dot) and the randomized placements for each netlist instance (files *_tr0000.ord). Instances of *.dot and *_tr0000.ord pairs serve as inputs to each subsequent treatment (and are downloaded by experimentalist one-by-one or as a compressed archive). Optimized solutions from new treatments (generated by experimentalists) are archived by archivist as files *_tr00xx.ord, where tr00xx designates the treatment number. Shown here are solutions for tr0012 and tr0030.

Each treated solution is evaluated for its cost independently by a common evaluator, creating a file tr00xx.tab. Statistical summaries for all instances and each treatment are stored in files tr00xx.sum. A grand statistical summary for all experimental treatments, generated using OmniFlows in Figures 5.7, 5.8, 5.9, and 5.10, is saved in the file allTreats.sum and also shown below.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Cost,P</th>
<th>Population Mean</th>
<th>Sample Mean</th>
<th>Sample Std. Dev.</th>
<th>Min.</th>
<th>Max.</th>
</tr>
</thead>
<tbody>
<tr>
<td>cn_tr0000</td>
<td>[1.197e+02, 1.306e+02]</td>
<td>1.2521e+02</td>
<td>1.5648e+01</td>
<td>4.5000e+01</td>
<td>1.3700e+02</td>
<td></td>
</tr>
<tr>
<td>cn_tr0012</td>
<td>[2.114e+01, 2.272e+01]</td>
<td>2.1937e+01</td>
<td>2.2134e+00</td>
<td>2.1000e+01</td>
<td>2.7000e+01</td>
<td></td>
</tr>
<tr>
<td>cn_tr0030</td>
<td>[2.986e+01, 3.138e+01]</td>
<td>3.0875e+01</td>
<td>2.8141e+00</td>
<td>2.5000e+01</td>
<td>3.6000e+01</td>
<td></td>
</tr>
</tbody>
</table>

Figure 5.11: Statistical summary of OpenExperiment by 5 participants on 4 hosts.

5.4 OpenWriter Taskflow Implementation

While this project could not be completed with the latest version of the OmniFlow, its initiation played an important role in devising the OmniFlow environment as presented in this thesis.

Major goals of OpenWriter include maintaining a collaborative project directory
for a book, an article, or a technical report that are to be compiled in LaTeX on a local hosts by individual contributors and synchronized on the collaboration server by the project editor and all participating writers. Utilities in this environment include support for hierarchical compilation of subsections, sections, and chapters; support for browsing the database on the server for citation keywords and for automated additions of citations to the database; support for file ownership, collaborative editing and compilation of any subsection, section, and chapter, of the project document.

The senior project during Summer 2000 created the basic utilities that were tested in the OmniFlowLite environment [115]. The environment itself is shown in Figure 5.12. Utilities are grouped in such a way, that a number of them can be chained for single-click execution or each can be executed as a stand-alone task in the OmniFlowLite environment.

Figure 5.12: Client/Server Utilities for a Collaborative LaTeX Environment.

There are a total of six taskflows, each flow corresponding to sequence of tasks that are repeatedly invoked. We next describe each flow in detail and the functionality of the various tasks within each flow:

**Flow 1.** This taskflow can execute, subject to user configuration of task connectors, all five tasks listed below. Its purpose is to allow the user to install files on the server
for further processing.

*Clean ServerDir:* This task removes a directory and a number of files from the server: (1) `path.tar`, where the path is a user-specified variable containing the name of the directory in which the files have been unpacked, and (2) `rootName *`, where `rootName` is a user-specified variable denoting the root name of the latex document.

*Upload LaTeX files:* This task simply uploads a file, `rootName.tar.gz`, where `rootName` is a user-specified variable, from the local directory to the server.

*Unpack files:* This task gunzips, untans, and copies the files up to the current working directory, effectively installing the files on the server.

*Make executable:* This task turns on the execute bit of the tcl scripts needed for execution and maintenance of the citation database.

*Generate Lib Index:* This task produces an index of tcl scripts necessary for accessing the library files.

**Flow 2.** This taskflow consists of a single task which allows the user to search a database of citations.

*bibSearch:* This task invokes the bibSearch script. Before clicking on the task node to invoke it, user is expected to enter a value for a search string variable ‘bib-SearchPattern’ into the entry box at the top of the client. Optionally, user can also enter a value ‘-full’ for the variable ‘bibSearchFlag’. With this option, the result file ‘bibFile.out’ produces full length citations that contain the search string. Without the optional input, the result file ‘bibFile.out’ produces only the list of citation keys of bibliographical database entries that contain the search string.

**Flow 3.** This taskflow also consists of a single task which allows the user to create an index of citation keys.
makeBibIndex: This task invokes the makeBibIndex script. Before clicking on the
task node to invoke it, user is expected to enter a value for the variable ‘cen-
tralDB’ into the entry box at the top of the client. If user enters the name of
’someDB’, a database someDB.bib is expected to exist as an input file to the
task. The output file generated will be named someDB.keys and will contain
the list of all citation keys in the database, including the line number on which
the citation key was found.

Flow 4. This taskflow can execute, subject to user configuration of task connectors,
all four tasks listed below. The main purpose of this taskflow is to generate dynami-
cally the local database of citations from the central database residing on the server.
Any citations not found in the central database are created by the user automatically
and verified for correct syntax by invoking the last task in this flow, bibtex.

LaTeX: This task invokes the LaTeX compiler. Nominal inputs are source files of
type *.tex, referenced by user-defined main.tex, e.g. A00_main.tex. Nominal
outputs are files rootName.dvi and rootName.aux. Additional files may be gen-
erated such as rootName.toc (data to generate table of contents), rootName.lof
(data to generate list of figures), etc. The file rootName.dvi renders the type-
set pages, the file rootName.aux contains data about objects referenced in the
source files, including all citation keys. Each citation key, entered by the user in
the source file, references a citation entry in one of the accessible bibliography
database files. These files have the extension *.bib and are expected in the
document working directory.

aux2keys: This task invokes the aux2keys script. Nominal input is a user-defined
file rootName.aux, e.g. A00_main.aux. Nominal output is a file of type root-
Name.keys, e.g. A00_main.keys.

keys2bib: This task invokes the keys2bib script. Two input files are expected: user-
defined files *.keys and *.bib, e.g. A00_main.keys and A00_main_central.bib.
Two user-defined output files are generated: a file with extension *.bib and a file
of extension *.keys, e.g. A00_main_local.bib and A00_main_local.keys. The file A00_main_local.bib contains database entries extracted from A00_main_central.bib, given A00_main.keys as the input. Any citations not found in A00_main_central.bib are identified by keys returned in the file A00_main_local.keys.

bibtex: This task invokes the bibtex program. The input files expected are: single user-defined file of type *.aux (e.g. A00_main.tex, generated by invoking LaTeX on A00_main.tex) and any number of *.bib files, identified in A00_main.tex; e.g. A00_main_local.bib and Z_citationDB.bib. The latter file is generated by user manually. It uses citation keys produced by A00_main_local.keys (these are the keys of entries not found in the central database). If no syntax errors are reported by bibtex, the output files that are generated are of rootName.bbl and rootName.blg, e.g. A00_main.bbl and A00_main.blg.

Flow 5. This taskflow consists of a single task which allows the user to add a user-generated (and syntactically verified) citation database file to the central database residing on the server.

add2centralDB: This task invokes the add2centralDB script. Its nominal inputs are user-specified files $centralDB.bib $centralDB.key $localBibFile $localKeyFile $submitKeyFile where ‘$’ signifies that all rootnames are variables entered by the user. The output are safely augmented files $centralDB.bib and $centralDB.key. Not all entries from $localBibFile are submitted, only the ones specified by the user with the citation keys in the file $submitKeyFile.

Flow 6. This taskflow can execute, subject to user configuration of task connectors, all five tasks listed below. The main purpose of this taskflow is to complete the compilation of the LaTeX document, now that all citations have been entered and verified for syntax as per taskflows above.

LaTeX: This task consists of first invocation of LaTeX compiler. Nominal inputs are files of type rootName.tex, rootName.bbl, rootName.blg. The essential output is file type rootName.dvi.
Latex: This task consists of second invocation of Latex compiler. Nominal inputs are files of type rootName.tex, rootName.bbl, rootName.blg. The essential output is file type rootName.dvi. Indices for all citations, figures and tables are expected be correctly cross-referenced after the second invocation.

dvips: This task invokes the dvips utility program. Nominal input is a file of type rootName.dvi, e.g. A00_main.dvi. Nominal outputs is a file to type rootName.ps, e.g. A00_main.ps.

ps2pdf: This task invokes the ps2pdf utility program. Nominal input is a file of type rootName.ps, e.g. A00_main.ps. Nominal outputs is a file to type rootName.pdf, e.g. A00_main.pdf.

latex2html: This task invokes the latex2html utility program. Nominal inputs are a number of files used to compile a latex document. Nominal output is a directory with name rootName-html which contains files with extension *.html and *.gif, producing a hypertext-linked version of the latex document.

OpenWriter execution. The execution of the OpenWriter taskflow proceeded in the order of the description above. Specifically, input data was designed such that on the first pass, the taskflows 1–3 executed successfully, whereas the taskflow 4 failed. The reason that the task ‘bibtex’ in the Flow 4 failed, was that not all citation keys generated as the output of the task ‘aux2keys’ were present in the ‘A00_main_central.bib’ when searched by the task ‘keys2bib’ that generated a ‘A00_main_local.bib’ as the input to task ‘bibtex’. Since ‘bibtex’ also reads the file ‘A00_main.aux’, it detects the missing keys and aborts before completion. However, the same missing keys were also detected and saved in the file A00_main_missing.keys by the task ‘keys2bib).

It is now up to the user to create bibliographical citations for the missing keys and save them in the file A00_tmp_new.bib. Upon creation of this file, the execution proceeds and completes successfully as follows:

- copy the existing A00_main.tex to A00_tmp.tex
• include the new citation file in A00_.tmp.tex under following syntax:
  \bibliography{A00_main_local,A00_tmp_new}
• re-execute the Flow 4 under the rootname A00_.tmp. If there are no syntax errors in A00_.tmp_new.bib, the task ‘bibtex’ should complete.

Having completed Flow 4, the user can now be confident that the citation entries in A00_.tmp_new.bib have no syntax errors and can be safely merged with the ’centralDB.bib’, using Flow 5. The Flow 6, having generated all the citation indices, and if without syntax errors of other kind, should now execute successfully for all tasks in the flow.

The role of OmniFlow (latest version). The taskflows for the OpenWriter project, built using the OmniFlowLite version, did have few limitations. It was very specific to single directory for latex compilation and did not allow the user to easily use the same flow recursively through a hierarchy of sub-directories. This required that the entire latex document reside in a single directory. Another limitation of OmniFlowLite is the absence of hierarchical flows due to which it becomes difficult to build and manage complex flows containing large number of task nodes in it.

We intend to re-implement the OpenWriter project using the latest version of OmniFlow that supports not only hierarchical taskflows but will also allow us to organize the directory structure of the latex document into several smaller manageable sub-directories of latex documents.

5.5 Summary

Three major groups of experiments were devised with OmniFlow to demonstrate its utility and versatility:

• OmniFlow Scalability reveals an exceptionally good asymptotic performance of the task scheduler – lending support for the efficiency of the scheduling algorithm and its implementation.
• *Distributed OpenDesign Environment* allows us to mix and match commercial and university-based tool to create a project-specific VLSI design flow environment.

• *Distributed OpenExperiment Environment* demonstrates not only the principles of a collaborative distributed computing environment but also specific role modes such as Archivist, Librarian, and Experimentalist.

Due to limitations of time, the scale of OpenDesign and OpenExperiment environment projects presented in this thesis is relatively small. However, the concepts introduced and verified can readily scale to large environments as demonstrated in the scalability experiments.

The OpenWriter environment, while having an important role in devising the presented OmniFlow environment, has been completed only in part and requires additional work in a future project.
Chapter 6

Conclusions and Future Work

This thesis makes a contribution towards a powerful distributed taskflow architecture that supports concurrency, with a highly interactive GUI-based taskflow design and execution environment, and an effective computing environment that facilitates asynchronous as well synchronous modes of collaboration. We first present a summary and conclusions of this thesis, followed by a brief overview on direction for future work.

6.1 Summary and Conclusions

We have developed the taskflow architecture to support the four basic structures, required to write any program as a hierarchy of tasks: (1) task sequencing, (2) decisions, allowing data to control the task sequence, (3) iterations, repeating the same task sequence a number of times, and (4) encapsulation, replacing a group of tasks with a name that denotes the group. The design and development of taskflows in our environment, called OmniFlows, allows one to encapsulate any tool that can be represented as a blackbox component. A blackbox component may represent a wide variety of tools, ranging from legacy applications to newly written programs, a generic cgi-script accessible on a web server to highly customized application specific services, such as JavaCADD.

The design methodology proposed in this thesis for creation of taskflows allows one
to leverage the benefits of concurrency very naturally, without introducing the complexity of threaded programming. Each task is always bounded by a ControlJoin and a ControlFork primitive on its input and its output respectively, so synchronization of its predecessor tasks and data-dependent selective invocation of its successor tasks occurs at each node. Therefore all tasks can be essentially executed concurrently by merely specifying the list of predecessor and successor tasks.

In the taskflow architecture, we define and make use of whitebox components, that support a highly interactive GUI-based taskflow design and execution environment. Additionally, we have introduced a data multiplexer primitive that facilitates one to repeatedly invoke a taskflow with each task executing either in a stand-alone mode with local data only, or in the context of the entire taskflow with the flow data. Additional user-control edges provide finer control over which tasks need to executed, either with local data or with flow data.

An environment for asynchronous collaboration is created by assignment of ownership of tasks and data in an OmniFlow to participating team members in a project. A asynchronous group client/server manages the information necessary to coordinate the authorized execution of tasks and prevent read/write conflicts arising due to shared data.

An environment for synchronous collaboration facilitates negotiated sharing of task sequencing and interactive execution of one or more of a sequence of tasks and taskflows. We have developed a novel hybrid collaboration architecture to render any Tk-based, single-user application collaborative without a code re-write. Additionally, configuration files for inter-client synchronization can be readily created to render a single-user application into a number of different collaboration modes. The evaluation of the collaboration toolkit shows that it has a good level of support for collaboration parameters and with minimal amount of programming effort required.

We have also developed WebWiseTclTk, an enhancement to the safe-tcl plugin that allows us to render tcl/tk applications inside a web browser without compromising security. Experimental setup of Tk widget demos as a testbed indicate that the WebWiseTclTk environment successfully passed 91 test cases out of a total of 104 test cases. The remaining 13 test cases failed because it would otherwise result in
compromise of the security policy.

Finally, we have defined an XML schema specification for collaborative distributed tasks mark-up language (cdtML) that allows us to create taskflows using XML format. This has the benefit of achieving language independence of the environment to any specific programming or scripting languages such as Tcl, or Perl. Additionally, wide availability of XML editors, validators and parser makes XML the best choice for specification of taskflows.

The OmniFlow scalability experiment, consisting of up to 9150 task instances, reveals an exceptionally good asymptotic performance of the task scheduler – lending support for the efficiency of the scheduling algorithm and its implementation. On the other hand the distributed OpenDesign and OpenExperiment projects, devised with OmniFlows, demonstrates the utility and versatility of the OmniFlow environment by encapsulating a wide variety of tools, including commercial and university-based tools.

6.2 Directions for Future Work

We conclude this thesis with a note about future work. Some may give additional support to the concepts presented in this thesis, other work may lead to significantly different contributions. The future work can be both theoretical and experimental. In the theoretical domain, the following problems are worth considering:

- It will be worthwhile to extend the formalization of the control/data taskflow model. The extended model would be useful in better analysis of taskflow structures and may allow us to include more advanced workflow patterns.

- Another important topic for future work is to remove the current limitation of requiring a reset signal from the user for repeated execution of the taskflow. This topic has the potential of allowing pipeling of task invocations and thereby improving its overall efficiency.

- Yet another topic for future work is to support optimization of resource utilization and perform appropriate load balancing. This topic will allow us to
optimize large complex taskflows during execution.

In the experimental domain, the major work that needs to be done is to widely deploy the OmniDesk architecture and implement a number of experimental projects for a wider range of applications. Elaborate experiments, similar to the ones demonstrated in Chapter 5 for the OmniFlow environment, need to be designed to determine the scalability of the OmniDesk environment.

Finally, a concise user-guide that introduces the OmniFlow/OmniDesk from the user perspective, along with several more examples of cdML specifications of projects will be needed to train new users.
Bibliography


A reprint also accessible from http://www.cbl.ncsu.edu/publications/-#I999-ISCAS-Stallmann.


#2000-TR@CBL-01-Ghosh.

#2000-TR@CBL-04-Stallmann.


[34] F. Bringlez. Frontiers of Collaborative Computing on the Internet, A Graduate Course Experiment, January 1999. Two project reports, published after the completion of the course, are also available from the course home page under http://www.cbl.ncsu.edu/~bringlez/csc591b/.


[40] Ada Resources. For more information, see http://1glwww.epfl.ch/Ada.


[86] Verilog Resources Home Page. For more information, see http://www.verilog.com/.

[87] VHDL Resource Page. For more information, see http://www.eda.ei.tum.de/forschung/vhdl.


[95] CORBA Resources Home Page. For more information, see http://www.corba.org/.
[96] JavaRMI Resources Home Page. For more information, see http://java.sun.com/marketing/collateral/javarmi.html.

[97] Simple Object Access Protocol (SOAP), 2000. For more information, see http://www.w3.org/TR/SOAP.

[98] Seventh International Symposium on Asynchronous Circuits and Systems. For more information, see http://www.async.elen.utah.edu/~async0.

[99] Component Pascal Resources. For more information, see http://www.oberon.ch.


[113] Home page of OpenProjects at CBL. For more information, see http://www.cbl.ncsu.edu/OpenProjects.


[117] Impress Home Page. For more information, see http://www.ntlug.org/~ccox/impress.

[119] Experimental Design Tutorial. The University of Oklahoma, Department of Family & Preventive Medicine, March 2000. For more information, see http://www.fammed.uohsc.edu/TUTOR/expdes.htm.


Appendix A

Definitions of a FSM and a FSMD

A finite-state machine (FSM) and a finite-state machine with a data path (FSMD) are one of the most popular design models in high-level synthesis and design of hardware systems. The formal definitions of both of these models are given in [83]. Here we briefly describe the definitions of both of these models.

A.1 FSM Model

The FSM model consists of a set of states, a set of transitions between states, and a set of actions associated with states, transitions or both states and transitions. More formally, a FSM is a quintuple

\[ < S, I, O, f : S \times I \to S, h : S \times I \to O > \]

where \( S = \{ s_i \} \) is a set of states, \( I = \{ i_j \} \) is a set of input values, and \( O = \{ o_k \} \) is a set of output value; \( f \) and \( h \) are next-state and output functions that map a cross product of \( S \) and \( I \) into \( S \) and \( O \), respectively.

An autonomous FSM is obtained if the input set \( I \) is empty and is used for components such as modulo-n counters. Here the next-state and output functions \( f \) and \( h \) are defined as mappings \( S \to S \) and \( S \to O \). On the other hand, both state-based and transition-based FSMs have a nonempty input set \( I \) and differ in the specification of the output function \( h \). A state-based FSM is called a Moore machine...
where the output value depends only on the state of the FSM \((h : S \rightarrow O)\). A transition-based FSM is called a *Mealy* machine where the output value depends on the state as well as input values \((h : S \times I \rightarrow O)\). In a transition-based FSM, the output will change only when input value changes, while the state will change only on the next clock pulse. In a state-based FSM, the output will persist until the state changes, independent of when the input value changes.

### A.2 FSMD Model

The FSM model works well for a few to several hundred states. Beyond several hundred states, the model becomes incomprehensible. In order to make FSM model usable for complex designs, a set of integer and floating-point variables are introduced where each variable represents thousands of different states. For example a 16-bit integer variable represents \(2^{16}\) or 65536 different states. Thus the introduction of a 16-bit variable reduces the number of states in the FSM model by 65536. Such a use of variables leads to the concept of an FSM with a data path (FSMD).

Let \(VAR\) represent a set of storage variables, \(EXP = \{f(x, y, z, \ldots) | x, y, z, \ldots \in VAR\}\) represent a set of expressions, and \(A = \{X \leftarrow e | X \in VAR, e \in EXP\}\) represent a set of storage assignments. Furthermore, let \(STAT = \{Rel(a, b) | a, b \in EXP\}\) represent a set of status signals as logical relations between the two expressions from the set \(EXP\). Given these definitions, an FSMD can be defined as the quintuple

\[
< S, I \cup STAT, O \cup A, f, h >
\]

where \(S\), \(f\), and \(h\) are defined as before, the set of input values is extended to include a combination of status values, and the output set is extended to include storage variable assignments.

In the context of our work, the *BlackBox* components forms an extension of the data path and communicates with the FSM by way of two handshaking signals. Section 3.3 discusses in detail the use of FSMD model for the taskflow architecture and its scheduling.
Appendix B

Specification of Cdtml Schema in XML

The XML schema for collaborative distributed tasks markup-language (Cdtml), as defined in this thesis (Chapter 3, Figure 3.17), is listed here. This schema conforms to the latest XML Schema Candidate Recommendation released by the W3C on 10/24/2000.

```xml
<?xml version="1.0"?>
<!-- cdtml_schema.xml Begin -->
<xsd:schema
 xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
 xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
 xmlns:dt="urn:W3C.org:xmldatatypes">

<!-- declare a "Text" element which is equivalent to PCDATA in DTDs -->
<xsd:complexType name="Text">
   <xsd:simpleContent>
      <xsd:restriction base="xsd:string"/>
   </xsd:simpleContent>
</xsd:complexType>

<!-- declare a "Url" type which is equivalent to href for html -->
<xsd:simpleType name="Url">
   <xsd:restriction base="xsd:uriReference">
      <xsd:minLength value="0"/>
      <xsd:maxLength value="unbounded"/>
   </xsd:restriction>
</xsd:simpleType>

<!-- declare a "InstanceName" type and a "InstanceList" type which -->
specifies the restrictions on instance names/list of a task -->
<xsd:simpleType name="InstanceName">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="3"/>
    <xsd:maxLength value="umbounded"/>
    <xsd:pattern value="\\([^\(\)]+)\)/">
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="InstanceList">
  <xsd:list itemType="InstanceName"/>
</xsd:simpleType>

<!-- declare a "InstanceState" type which enumerates the various
instance states of a control port -->
<xsd:simpleType name="InstanceState">
  <xsd:restriction base="xsd:string">
    <xsd:minLength value="1"/>
    <xsd:maxLength value="2"/>
    <xsd:pattern value="\\([EXTVIA]\\|A\[up\\])"/>
  </xsd:restriction>
</xsd:simpleType>

<!-- top-level element for invoking cdml files -->
<xsd:element name="Cdtml">
  <xsd:complexType>
    <xsd:choice maxOccurs="umbounded">
      <xsd:element name="SingleTaskDefn" type="SingleTaskDefn"/>
      <xsd:element name="SingleTaskBody" type="SingleTaskBody"/>
      <xsd:element name="MultiTaskDefn" type="MultiTaskDefn"/>
      <xsd:element name="MultiTaskBody" type="MultiTaskBody"/>
      <xsd:element name="TaskInstance" type="TaskInstance"/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>

<!-- element for task instance -->
<xsd:complexType name="TaskInstance">
  <xsd:sequence>
    <xsd:element name="ControlJoin" type="ControlJoin" minOccurs="0"/>
    <xsd:element name="DataMux" type="DataMux"/>
    <xsd:element name="SkipCondition" type="CtrlCondition" minOccurs="0"/>
    <xsd:choice>
      <xsd:element name="SingleTaskDefn" type="SingleTaskDefn"/>
      <xsd:element name="MultiTaskDefn" type="MultiTaskDefn"/>
    </xsd:choice>
    <xsd:element name="RepeatCondition" type="CtrlCondition" minOccurs="0"/>
    <xsd:element name="ControlFork" type="ControlFork" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:attribute name="instance">
  <xsd:simpleType>
    <xsd:restriction base="InstanceName"/>
  </xsd:simpleType>
</xsd:attribute>

<xsd:attribute name="host">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>
</xsd:attribute>

<xsd:attribute name="workdir">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>
</xsd:attribute>

<xsd:attribute name="owner">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>
</xsd:attribute>

</xsd:complexType>

<!-- type declaration of single task definition -->
<xsd:complexType name="SingleTaskDefn">
  <xsd:sequence minOccurs="0">
    <xsd:element name="Title" type="Title"/>
    <xsd:element name="Description" type="Description"/>
    <xsd:element name="InputList" type="InputList"/>
    <xsd:element name="InOutList" type="InOutList"/>
    <xsd:element name="OutInList" type="OutInList"/>
    <xsd:element name="OutputList" type="OutputList"/>
    <xsd:element name="ExtTask" type="ExtTask"/>
  </xsd:sequence>

  <xsd:attribute name="name">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>

<!-- type declaration of single task body -->
<xsd:complexType name="SingleTaskBody">
  <xsd:sequence>
    <xsd:element name="BeginCondition" type="CtrlCondition" minOccurs="0"/>
    <xsd:element name="InvokeCommand" type="InvokeCommand"/>
    <xsd:element name="EndCondition" type="CtrlCondition" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType>
  <xsd:restriction base="xsd:string"/>
</xsd:complexType>

<!-- type declaration of multi task definition -->
<xsd:complexType name="MultiTaskDefn">
  <xsd:sequence minOccurs="0">
    <xsd:element name="Title" type="Title"/>
    <xsd:element name="Description" type="Description"/>
    <xsd:element name="InputList" type="InputList"/>
    <xsd:element name="InOutList" type="InOutList"/>
    <xsd:element name="OutInList" type="OutInList"/>
    <xsd:element name="OutputList" type="OutputList"/>
    <xsd:element name="TaskList" type="TaskList"/>
    <xsd:element name="TaskGraph">
      <xsd:complexType>
        <xsd:sequence/>
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
</xsd:complexType>

<!-- type declaration of multi task body -->
<xsd:complexType name="MultiTaskBody">
  <xsd:sequence>
    <xsd:element name="BeginCondition" type="CtrlCondition" minOccurs="0"/>
    <xsd:element name="Begin" type="ControlFork"/>
    <xsd:element name="TaskInstance" type="TaskInstance"
      maxOccurs="unbounded"/>
    <xsd:element name="End" type="ControlJoin"/>
    <xsd:element name="EndCondition" type="CtrlCondition" minOccurs="0"/>
    <xsd:element name="DataGraph" minOccurs="0"/>
  </xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:sequence>

<xsd:attribute name="name">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string"/>
  </xsd:simpleType>
</xsd:attribute>
</xsd:attribute>
</xsd:complexType>

<!-- type declaration of external command line -->
<xsd:complexType name="InvokeCommand">
  <xsd:attribute name="name">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
  </xsd:attribute>
  <xsd:attribute name="defaulthost">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
  </xsd:attribute>
  <xsd:attribute name="defaultworkdir">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
  </xsd:attribute>
  <xsd:attribute name="defaultowner">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
  </xsd:attribute>
  <!-- default owner is "nobody" -->
</xsd:complexType>

<!-- type declaration of ControlFork -->
<xsd:complexType name="ControlFork">
  <xsd:sequence>
    <xsd:element name="NextTask" type="CtrlCondition"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<!-- type declaration of ControlJoin -->
<xsd:complexType name="ControlJoin">
  <xsd:sequence>
    <xsd:element name="PrevTask" type="CtrlCondition"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<!-- type declaration of CtrlCondition -->
<xsd:complexType name="CtrlCondition">
  <xsd:attribute name="instanceEnable">
    <xsd:simpleType>
      <xsd:restriction base="InstanceName"/>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>
<!-- type declaration of data multiplexer -->
<xsd:complexType name="DataMux">
  <xsd:sequence>
    <xsd:element name="SetLocal" type="SetPort"
      minOccurs="0" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<!-- type declaration of data fork -->
<xsd:complexType name="DataFork"/>

<!-- type declaration of data selector -->
<xsd:complexType name="DataSelector"/>

<!-- type declaration of data collector -->
<xsd:complexType name="DataCollector"/>

<!-- declare a PortName type which can be used by input, inout and output elements -->
<xsd:complexType name="PortName">
  <xsd:sequence>
    <xsd:element name="Title"/>
    <xsd:element name="SetDefault" type="SetPort"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="InputList">
  <xsd:sequence minOccur="0" maxOccur="unbounded">
    <xsd:element name="Input" type="PortName"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="InOutList">
  <xsd:sequence minOccur="0" maxOccur="unbounded">
<xsd:element name="InOut" type="PortName"/>
</xsd:sequence>
</xsd:complexType>
<xsd:complexType name="OutInList">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="OutIn" type="PortName"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="OutputList">
  <xsd:sequence minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="Output" type="PortName"/>
  </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="ExtTask">
  <xsd:sequence>
    <xsd:element name="ProgramName" type="PortName"/>
  </xsd:sequence>
</xsd:complexType>

<!-- type declaration for TaskList and Task -->
<xsd:complexType name="TaskList">
  <xsd:sequence>
    <xsd:element name="Begin" type="BeginOrEndNode"/>
    <xsd:element name="Task" type="TaskNode" maxOccurs="unbounded"/>
    <xsd:element name="End" type="BeginOrEndNode"/>
  </xsd:sequence>
</xsd:complexType>

<!-- type declaration of BeginOrEndNode -->
<xsd:complexType name="BeginOrEndNode">
  <xsd:attribute name="level">
    <xsd:simpleType>
      <xsd:restriction base="xsd:integer"/>
    </xsd:simpleType>
  </xsd:attribute>
  <xsd:attribute name="offset">
    <xsd:simpleType>
      <xsd:restriction base="xsd:integer"/>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>

<!-- type declaration of TaskNode -->
<xsd:complexType name="TaskNode">
  <xsd:attribute name="instance">
    <xsd:simpleType>
      <xsd:restriction base="InstanceName"/>
    </xsd:simpleType>
  </xsd:attribute>
  <xsd:attribute name="taskref"/>
<xsd:simpleType>
  <xsd:restriction base="Url"/>
</xsd:simpleType>
</xsd:complexType>

<!-- type declaration for SetPort -->
<xsd:complexType name="SetPort">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="Value"/>
    <xsd:element name="File"/>
  </xsd:choice>
  <xsd:attribute name="name">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>

<!-- type declaration for GetPort -->
<xsd:complexType name="GetPort">
  <xsd:choice minOccurs="0" maxOccurs="unbounded">
    <xsd:element name="Value"/>
    <xsd:element name="File"/>
  </xsd:choice>
  <xsd:attribute name="port">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string"/>
    </xsd:simpleType>
  </xsd:attribute>
</xsd:complexType>

<!-- type declaration for Title -->
<xsd:complexType name="Title"/>

<!-- type declaration for Description -->
<xsd:complexType name="Description"/>

</xsd:schema>
<!-- cdtml_schema.xml End -->
Appendix C

Taskflow Programming in Cdtml

The specification of taskflows involves creation of three types of cdtML files: (1) a main invocation file, (2) a task definition file, and (3) a task body file. The first two types of files are relatively simple to create since the main invocation file merely assigns data to ports of the main task instance, and the definition file consists of specifying a list of input, inout, outin and output data ports for a task and a list of task instances with its task-graph. On the other hand, creation of the third type of file, which is task body, is a bit more involved and can be considered to require some programming effort.

Here, we list and explain the syntax of the various commands that is typically used in the creation of the taskflow body. For the sake of completeness, we also explain the syntax of TaskGraph used in creation of the taskflow definition file.

TaskGraph: The structure of the TaskGraph element is similar to the Dot format description [124] and its syntax has been adapted for our environment. A TaskGraph specification consists of one or more pairs of task instances connected either with a control-edge or an abort-edge where all the three elements are separated by whitespace. The control-edge is represented by an equal sign ‘=’ and the abort-edge is represented by a plus sign ‘+’. With two task instances, say (T1) and (T2), there are four possibilities to form a TaskGraph since each edge can be either configured to be an open link or a closed link:
1. \((T1) \Rightarrow (T2)\) (closed control-edge)
2. \((T1) \equiv (T2)\) (open control-edge)
3. \((T1) \Rightarrow (T2)\) (closed abort-edge)
4. \((T1) \Leftrightarrow (T2)\) (open abort-edge)

Users can specify a TaskGraph of any complexity at a given level of hierarchy by merely connecting the list of task instances with appropriate edge connectors. The only restriction is that the two reserved key words (BEGIN) and (END), which signify the entry point and the exit point of the taskflow respectively, must be specified in a TaskGraph. Also, task repetition can be specified by connecting a task instance to itself using a control-edge. For example, the TaskGraphs \((T1) \Rightarrow (T1)\) and \((T1) \equiv (T1)\) both imply task repetition with a closed feedback link and an open feedback link respectively.

DataGraph: The structure of the DataGraph element is also similar to the Dot format description [124]. A DataGraph specification consists of one or more pairs of port names of task instances connected with a data-to-data edge and is represented by ‘->’. Here again the three elements are separated by whitespace. Since, at a given level of hierarchy of a multi-task definition, two or more of its task instances may have a port by same name, it is essential to qualify every port name with its task instance name, separated by a period. The only exception is that the ports of the multi-task body, containing the DataGraph, should not be specified with a qualifying port name. The example below shows one such declaration of a DataGraph.

Consider three task instances \((T1), (T2)\) and \((T3)\) with one input port \(m\text{In}\) and one output port \(m\text{Out}\) each. Also, let \((T1)\) be a hierarchical task which encapsulates the remaining two tasks. A typical DataGraph for such a taskflow would be as follows:

\[
\begin{align*}
m\text{In} & \quad \rightarrow \quad (T2).m\text{In} \\
(T2).m\text{Out} & \quad \rightarrow \quad (T3).m\text{In}
\end{align*}
\]
(T3).mOut -> mOut

Here, the input port mIn of hierarchical task (T1) is connected to input port mIn of task instance (T2) on the first line. The second line connects the output of (T2) to input of (T3) and the third line connects the output of (T3) to output of the hierarchical task (T1). Note that the input and output ports of task (T1) are not specified with a qualifying task name preceding it.

Attributes of task instance: A task instance currently has five attributes, whose values can be assigned, when the task instance is encapsulated in the body of a multi-task. The list five attributes are described below:

sleep: The sleep attribute is used during the simulation of taskflow. Its value is specified as an integer number (in seconds) which is used to specify the amount of time the corresponding task instance sleeps for during simulation. If no value is specified for this attribute, then it internally generates a random number in the range of 0 to 5 seconds with a precision of one milli-second during simulation.

host: The host attribute is used to designate a task instance on a specific host for execution. If no host attribute is specified then its value defaults to 'localhost'.

workdir: The workdir attribute is used to specify the directory in which the task instance needs to be executed. If no value is specified for workdir attribute, it executes the task in the current invocation directory.

owner: The owner attribute is used to designate a user as the owner of the specified task instance. In the absence of an owner attribute, the default owner of a task is the current user who has invoked the task instance.

maxIterate: The maxIterate attribute is useful only for tasks which are repeatedly invoked and specifies a maximum count of how many times the corresponding task instance may be repeated. In the absence of this attribute, the default maximum limit of task repetition is 1000.
Access to data port: In the body of a multi-task or a single-task, it may be necessary to access the value of a data port. This is necessary so as to invoke the blackbox component with correct data value or to derive the name of one data port based on the value of another data port. Accordingly, we provide four commands that may be used to access the value of a data port and one command to access the value of an attribute of a task instance. The syntax of these commands is as shown below:

\[
\begin{align*}
\text{cdtGetData} & \quad \text{portname} \\
\text{cdtReadData} & \quad \text{portname} \\
\text{cdtSetData} & \quad \text{portname} \quad \text{value} \\
\text{cdtWriteData} & \quad \text{portname} \quad \text{contents} \\
\text{cdtGetAttr} & \quad \text{attrname}
\end{align*}
\]

The first command `cdtGetData` expects the name of the port as its argument and returns its corresponding data value. The returned value will be the filename, if the port is of persistent type, otherwise it returns the actual data value. The second command `cdtReadData` expects the name of the port whose type is persistent only and it returns the contents of the corresponding filename. The third command `cdtSetData` expects two arguments, the name of the port and the value to be assigned to it. Similarly, the fourth command `cdtWriteData` expects two arguments where the first argument is the name of the port of type persistent and the second argument is the contents of the file that need to be saved in a filename. Finally, the fifth command `cdtGetAttr` is similar to `cdtGetData` and expects the name of an attribute of a task instance and returns its corresponding value.

Evaluation of control conditions: These are specified as part of ControlJoin and ControlFork primitives and also for the RepeatCondition. The syntax of the various types of conditions is show below:

\[
\begin{align*}
\text{evalJoinCondition} & \quad \text{boolean_opr} \quad ?\text{number} \quad \text{arg1} \quad \text{arg2} \ldots \quad \text{argN} \\
\text{evalAbortCondition} & \quad \text{boolean_opr} \quad ?\text{number} \quad \text{arg1} \quad \text{arg2} \ldots \quad \text{argN}
\end{align*}
\]
evalForkCondition successor_task value1 comparator value2
evalRepeatCondition value1 comparator value2

where,
  boolean_opr = And, Or, Not, Minimum.
  comparator = IsGreaterThan, IsGreaterEqual,
                IsLessThan,  IsLessEqual,
                IsEqual,     IsNotEqual.
  argi       = \{predecessor_task state\}

The evalJoinCondition is used to determine when the corresponding task instance should be enabled, skipped or aborted. It consists of a boolean operator whose value is one of And, Or Not and Minimum, followed by a number of arguments where each argument is a pair consisting of the name of the predecessor task and its corresponding state. The first three types of boolean operator are simple whereas the last type of operator specifies a minimum number of arguments that need to be valid for the join condition to be true. When the result of evaluating the join condition is true, it changes the corresponding task instance into enabled or skipped state depending on whether the evalJoinCondition is a part of UserInvoke or UserSkip element. The evalAbortCondition is very similar to the evalJoinCondition except for the difference that it aborts the corresponding task instance when the condition is evaluated to true.

On the other hand, the evalForkCondition is used to determine and sets the state of the corresponding control-edge or the abort-edge connecting the successor task. It has exactly four arguments, with the first argument specifying the name of the successor task, the second and the fourth argument are data values and the third argument is a comparator to compare the two data values. The two data values may be specified either as a literal string or one may use the cdtGetData and the cdtReadData to access the values for a specific port. The result of evaluation of the fork condition sets the corresponding status to either Valid or Invalid.

The evalRepeatCondition is very similar to the evalForkCondition and the
only difference between the two is that it requires only three arguments - the name of the successor task is not needed because it invokes itself.

**Encapsulation of blackbox component:** The blackbox components can consist of wide range of applications, ranging from simple utility programs to special executable programs, from cgi-scripts to customized applications. Most blackbox components can be invoked by invoking a process using the command-line invocation method. In addition, we also provide a set of library functions to encapsulate special-purpose types of blackbox components. Currently, this library is very small but is expected to grow as more and more tasks are encapsulated for various projects.

The following three commands show the functionality to encapsulate Internet-based components such as downloading Web data, invocation of cgi-scripts, and sending an e-mail.

```
  cdtHttpGet     url  ? local_file ?
  cdtHttpPost   url  parameter_list  ? local_file ?
  cdtSsmtpSend   smtp_host recipients from subject body
```

The first command `cdtHttpGet` is used to download the data located at the specified `url` and save it in `local_file`. The second command `cdtHttpPost` is used to invoke a cgi-script where the first argument is the `url` for the cgi-script, the second argument is list of parameters that need to be submitted to thecgi-script and the last argument specifies the local filename where the results of invoking the cgi-script should be saved. Finally, the third command `cdtSsmtpSend` can be used to send an e-mail, from a user-specified `smtp_host`, to a list of `recipients` with appropriate `subject` and the `body` matter.

The examples on usage of most of these commands are found in the parabola taskflow whose entire cdtML description is listed in Appendix D.
Appendix D

Cdtml Example: Parabola

Taskflow

The Cdtml specification of the ‘parabola’ taskflow is listed here. It consist of three files: (1) a main file specifies the list of task instances that are invoked at the toplevel, (2) a definition file specifies the API for single and multi task components, and (3) a body file contains all the details of various single and multi task components. Specifically, the parabola taskflow contains the following elements:

- one MainTask invocation,
- five single-task definitions: pInitSolver, pSolver, download, pEvaluator, and pReport,
- two multi-task definitions: task_parabola and pSolverFlow, and
- a total of fifteen task instances of single and multi task definitions.

The tree hierarchy of all the fifteen task instances of the parabola OmniFlow is shown below:

```plaintext
+ (parabola)  # task_parabola (multi-task)
  - (InitA)   # pInitSolver (single-task)
  + (A)       # pSolverFlow (multi-task)
    - (pSolverWeb) # pSolver (single-task)
    - (download)  # download (single-task)
    - (InitB)     # pInitSolver (single-task)
```
The taskflow is introduced in Section 3.1, Figure 3.4, and its GUI screenshot is shown in Section 3.5, Figure 3.23. We now look at the three files for the cdtML specification of the parabola OmniFlow. Note that in this example, all components are accessible as cgi-scripts on the web under:

http://www.cbl.ncsu.edu/vela/coPI-only/cgi-bin/pub/Parabola

Furthermore, the output data that is generated on execution of the ‘parabola’ taskflow is saved and also accessible on the web under:

http://www.cbl.ncsu.edu/vela/SavedData/lavana@cbl.ncsu.edu/Parabola

### D.1 Main Invocation

The MainTask invocation consist of only one task instance, called (parabola), which references the multi-task definition task_parabola in the definition file. In addition, the input and the output data ports of the task instance are initialized with appropriate data here.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CDTML SYSTEM "http://www.cbl.ncsu.edu/OpemCdt/ctml_schema.xml">

<!-- parabola_main.cdt -->
<CDTML>
    <MAINTASK>
        <TASKLIST>
            <BEGIN/>
            <TASK instance="(parabola)"
            taskRef="parabola_defn.cdt#task_parabola"/>
            <END/>
        </TASKLIST>
    </MAINTASK>
</CDTML>
```
D.2 Taskflow Definition

This file consists of five single-task definitions and two multi-task definitions. We will briefly describe the specification of the first multi-task definition task.parabola and the rest of the definitions are similar.

The definition of the multi-task task.parabola consists of specifying the attributes such as name of the multi-task under name= and a reference to the file containing the body of the multi-task under bodyRef=. In addition, the Title and the Description elements allow us to describe the functionality of the multi-task in brief and in detail, respectively.

Next the input/output ports for the multi-task are declared under InputList and OutputList elements. The TaskList element specifies the list of task instances invoked by the multi-task, where we assign a name for each task instance under
instance= and also specify the reference to the file containing the corresponding task
definition under taskRef=. Finally, the TaskGraph element specifies the dependencies
among the various tasks instances.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CDTML SYSTEM "http://www.cbl.ncsu.edu/OpenCdt/cdtm_schema.xml">

<!-- parabola_defn.cdt -->
<CDTML>

<MultiTaskDefn name="task_parabola" bodyRef="parabola_body.cdt#task_parabola">
  <Title> Taskflow task_parabola </Title>
  <Description> ... </Description>
  <InputList>
    <Input port="emailaddr" type="temporary">
      <Title> Email address </Title>
    </Input>
    <Input port="mInDescrA" type="persistent">
      <Title> Input description for task A </Title>
    </Input>
    <Input port="mInDescrB" type="persistent">
      <Title> Input description for task B </Title>
    </Input>
    <Input port="mInDescrC" type="persistent">
      <Title> Input description for task C </Title>
    </Input>
    <Input port="mInitCost" type="temporary">
      <Title> Initial cost (should be very large) </Title>
    </Input>
  </InputList>
  <OutputList>
    <Output port="mOutReport" type="persistent">
      <Title> Output report of parabolic evaluation </Title>
    </Output>
  </OutputList>
  <TaskList>
    <Begin/>
    <Task instance="(InitA)" taskRef="parabola_defn.cdt#pInitSolver"/>
    <Task instance="(InitB)" taskRef="parabola_defn.cdt#pInitSolver"/>
    <Task instance="(InitC)" taskRef="parabola_defn.cdt#pInitSolver"/>
    <Task instance="(A)" taskRef="parabola_defn.cdt#pSolverFlow"/>
    <Task instance="(B)" taskRef="parabola_defn.cdt#pSolverFlow"/>
    <Task instance="(C)" taskRef="parabola_defn.cdt#pSolverFlow"/>
    <Task instance="(D)" taskRef="parabola_defn.cdt#pEvaluator"/>
    <Task instance="(E)" taskRef="parabola_defn.cdt#pReport"/>
    <End/>
  </TaskList>
  <TaskGraph>
    (BEGIN) => (InitA) => (A) => (D) => (E) => (END)
  </TaskGraph>
</MultiTaskDefn>
</CDTML>
```
\[
\begin{align*}
(A) &\Rightarrow (A) \\
(BEGIN) &\Rightarrow (InitB) \Rightarrow (B) \Rightarrow (D) \Rightarrow (E) \Rightarrow (END) \\
(B) &\Rightarrow (B) \\
(BEGIN) &\Rightarrow (InitC) \Rightarrow (C) \Rightarrow (D) \Rightarrow (E) \Rightarrow (END) \\
(C) &\Rightarrow (C) \\
\end{align*}
\]

</MultiTaskDefn>

\[<\text{SingleTaskDefn name="pInitSolver" bodyRef="parabola_body.cdt#pInitSolver"}>
\]

\[<\text{Title}> \text{Initialize Parabola Solver } </\text{Title}>\]
\[<\text{Description}> ... </\text{Description}>\]
\[<\text{InputList}>\]
\[<\text{Input port="root" type="temporary"}>\]
\[<\text{Title}> Root name for the task instance </\text{Title}>\]
\[</\text{Input}>\]
\[<\text{Input port="inpDescr" type="persistent"}>\]
\[<\text{Title}> Description of input points </\text{Title}>\]
\[</\text{Input}>\]
\[<\text{Input port="initDataDir" type="temporary"}>\]
\[<\text{Title}> Initial data directory </\text{Title}>\]
\[<\text{DefaultValue}> initData </\text{DefaultValue}>\]
\[</\text{Input}>\]
\[</\text{InputList}>\]
\[<\text{InOutList}>\]
\[<\text{Input port="emailaddr" type="temporary"}>\]
\[<\text{Title}> Email address </\text{Title}>\]
\[</\text{Input}>\]
\[</\text{InOutList}>\]
\[</\text{SingleTaskDefn}>\]

\[<\text{MultiTaskDefn name="pSolverFlow" bodyRef="parabola_body.cdt#pSolverFlow"}>
\]

\[<\text{Title}> pSolver Flow </\text{Title}>\]
\[<\text{Description}> ... </\text{Description}>\]
\[<\text{InputList}>\]
\[<\text{Input port="emailaddr" type="temporary"}>\]
\[<\text{Title}> Email address </\text{Title}>\]
\[</\text{Input}>\]
\[<\text{Input port="root" type="temporary"}>\]
\[<\text{Title}> Root name for the task instance </\text{Title}>\]
\[</\text{Input}>\]
\[<\text{Input port="inpDescr" type="persistent"}>\]
\[<\text{Title}> Description of input points </\text{Title}>\]
\[</\text{Input}>\]
\[<\text{Input port="initCost" type="temporary"}>\]
\[<\text{Title}> Initial Cost </\text{Title}>\]
\[</\text{Input}>\]
\[<\text{Input port="initSolv" type="temporary"}>\]
\[<\text{Title}> Initial solution </\text{Title}>\]
\[</\text{Input}>\]
\[</\text{InputList}>\]
<OutInList>
  <OutIn port="newSoln" type="persistent">
    <Title> New Solution </Title>
  </OutIn>
  <OutIn port="newCost" type="persistent">
    <Title> New Cost </Title>
  </OutIn>
</OutInList>

<OutputList>
  <Output port="nIter" type="persistent">
    <Title> Number of Iterations </Title>
  </Output>
  <Output port="oldCost" type="persistent">
    <Title> Old Cost </Title>
  </Output>
</OutputList>

<TaskList>
  <Begin/>
  <Task instance="(pSolverWeb)" taskRef="parabola_defn.cdt#pSolver"/>
  <Task instance="(download)" taskRef="parabola_defn.cdt#download"/>
  <End/>
</TaskList>

<TaskGraph>
  (BEGIN) => (pSolverWeb) => (download) => (END)
</TaskGraph>

</MultiTaskDefn>

<SingleTaskDefn name="pSolver" bodyRef="parabola_body.cdt#pSolver">
  <Title> Parabola Solver </Title>
  <Description> ... </Description>

<InputList>
  <Input port="emailaddr" type="temporary">
    <Title> Email address </Title>
  </Input>
  <Input port="parabolaurl" type="temporary">
    <Title> Url for the parabola cgi-script </Title>
    <DefaultValue>
      http://www.cbl.ncsu.edu/vela/coPI-only/cgi-bin/pub/Parabola
    </DefaultValue>
  </Input>
  <Input port="root" type="temporary">
    <Title> Root name for the task instance </Title>
  </Input>
  <Input port="inpDescr" type="persistent">
    <Title> Description of input points </Title>
  </Input>
  <Input port="initCost" type="temporary">
    <Title> Initial Cost </Title>
  </Input>
  <Input port="initSoln" type="temporary">
    <Title> Initial Solution </Title>
  </Input>
</InputList>
<Title> Initial solution </Title>
</Input>
</InputList>
<OutInList>
  <OutIn port="newSoln" type="persistent">
    <Title> New Solution </Title>
  </OutIn>
  <OutIn port="newCost" type="persistent">
    <Title> New Cost </Title>
  </OutIn>
</OutInList>
</OutputList>
<OutputList>
  <Output port="results" type="persistent">
    <Title> Save results of the http submit </Title>
  </Output>
  <Output port="nIter" type="persistent">
    <Title> Number of Iterations </Title>
  </Output>
  <Output port="oldCost" type="persistent">
    <Title> Old Cost </Title>
  </Output>
</OutputList>
</SingleTaskDefn>

<SingleTaskDefn name="download" bodyRef="parabola_body.cdt#download">
  <Title> Download Files </Title>
  <Description> ... </Description>
  <InputList>
    <Input port="emailaddr" type="temporary">
      <Title> Email address </Title>
    </Input>
    <Input port="resultswrl" type="temporary">
      <Title> Url for the saved location of the results </Title>
      <DefaultValue>
        http://www.cbl.ncsu.edu/vela/SavedData/[cdtGetData emailaddr]/Parabola
      </DefaultValue>
    </Input>
    <Input port="newSolnWeb" type="persistent">
      <Title> New Solution </Title>
    </Input>
    <Input port="newCostWeb" type="persistent">
      <Title> New Cost </Title>
    </Input>
    <Input port="nIterWeb" type="persistent">
      <Title> Number of Iterations </Title>
    </Input>
    <Input port="oldCostWeb" type="persistent">
      <Title> Old Cost </Title>
    </Input>
  </InputList>
</SingleTaskDefn>
<OutputList>
  <Output port="newSolnLcl" type="persistent">
    <Title> New Solution </Title>
  </Output>
  <Output port="newCostLcl" type="persistent">
    <Title> New Cost </Title>
  </Output>
  <Output port="nIterLcl" type="persistent">
    <Title> Number of Iterations </Title>
  </Output>
  <Output port="oldCostLcl" type="persistent">
    <Title> Old Cost </Title>
  </Output>
</OutputList>

<SingleTaskDefn name="pEvaluator" bodyRef="parabola_body.cdt#pEvaluator">
  <Title> Parabola Evaluator </Title>
  <Description> ... </Description>
  <InputList>
    <Input port="costA" type="persistent">
      <Title> Cost of task A </Title>
    </Input>
    <Input port="costB" type="persistent">
      <Title> Cost of task B </Title>
    </Input>
    <Input port="costC" type="persistent">
      <Title> Cost of task C </Title>
    </Input>
  </InputList>
  <OutputList>
    <Output port="totalCost" type="persistent">
      <Title> Total cost </Title>
    </Output>
    <Output port="costAvg" type="persistent">
      <Title> Average cost </Title>
    </Output>
    <Output port="count" type="persistent">
      <Title> Total count of tasks evaluated </Title>
    </Output>
  </OutputList>
</SingleTaskDefn>

<SingleTaskDefn name="pReport" bodyRef="parabola_body.cdt#pReport">
  <Title> Parabola Report Generator </Title>
  <Description> ... </Description>
  <InputList>
    <Input port="totalCost" type="persistent">
      <Title> Total cost </Title>
    </Input>
  </InputList>
</SingleTaskDefn>
D.3 Taskflow Body

The taskflow body file consists of the body specification of the various tasks declared in the corresponding definition file. We next look at the body specification of the multi-task task_parabola.

In addition to the name= attribute, it may also contain attributes such as sleep= which specifies the time to be used during simulation of the taskflow environment. For each task instance contained in this multi-task, we specify an assignment of input/output data ports that can be used for invocation in local stand-alone mode. Additionally, we also specify the RepeatCondition, JoinCondition and ForkCondition conditions for the various task instances if required. In the body of task_parabola, a RepeatCondition is specified for the three task instances (A), (B), and (C) and a JoinCondition is specified for the task instance (D). Finally, the DataGraph element specifies the dependencies of the flow of data among various tasks.

The body of the single-task pSolver consists of encapsulation of the blackbox component. Here, the blackbox component is an invocation of a cgi-script accessible using the http protocol. Therefore, we use the ExecCommand element of type="HttpCmd" to encapsulate and invoke the cgi-script from within this task instance.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CDTML SYSTEM "http://www.cbl.ncsu.edu/OpenCdt/cdtm_schema.xml"
<MultiTaskBody name="task_parabola" sleep="1">
  <BeginFork/>
  <TaskInstance instance="(InitA)">
    <DataMux>
      <SetInput port="root">
        <LocalValue> taskA </LocalValue>
      </SetInput>
    </DataMux>
  </TaskInstance>
  <TaskInstance instance="(InitB)">
    <DataMux>
      <SetInput port="root">
        <LocalValue> taskB </LocalValue>
      </SetInput>
    </DataMux>
  </TaskInstance>
  <TaskInstance instance="(InitC)">
    <DataMux>
      <SetInput port="root">
        <LocalValue> taskC </LocalValue>
      </SetInput>
    </DataMux>
  </TaskInstance>

  <TaskInstance instance="(A)" maxIterate="15">
    <DataMux>
      <SetInput port="root">
        <LocalValue> taskA </LocalValue>
      </SetInput>
      <SetInput port="inpDescr">
        <LocalValue> taskA_inpDescr.d </LocalValue>
      </SetInput>
      <SetInput port="initCost">
        <LocalValue> 1.e-6 </LocalValue>
      </SetInput>
      <SetInput port="initSoln">
        <LocalValue> 5 </LocalValue>
      </SetInput>
      <SetOutIn port="newCost">
        <LocalValue> taskA_newCost.d </LocalValue>
      </SetOutIn>
      <SetOutIn port="newSoln">
        <LocalValue> taskA_newSoln.d </LocalValue>
      </SetOutIn>
      <SetOutput port="nIter">
        <LocalValue> taskA_nIter.d </LocalValue>
      </SetOutput>
    </DataMux>
  </TaskInstance>
</MultiTaskBody>
</SetOutput>
<SetOutput port="oldCost">
    <LocalValue> taskA_oldCost.d </LocalValue>
</SetOutput>
</DataMux>
<RepeatCondition>
    <UserRepeat>
        evalRepeatCondition \n        "[cdtReadData newCost]" IsLessThan "[cdtReadData oldCost]"
    </UserRepeat>
</RepeatCondition>
</TaskInstance>
<TaskInstance instance="(B)" maxIterate="15">
    <DataMux>
        <SetInput port="root">
            <LocalValue> taskB </LocalValue>
        </SetInput>
        <SetInput port="inpDescr">
            <LocalValue> taskB_inpDescr.d </LocalValue>
        </SetInput>
        <SetInput port="initCost">
            <LocalValue> 1.e6 </LocalValue>
        </SetInput>
        <SetInput port="initSoln">
            <LocalValue> 17 </LocalValue>
        </SetInput>
        <SetOutIn port="newCost">
            <LocalValue> taskB_newCost.d </LocalValue>
        </SetOutIn>
        <SetOutIn port="newSoln">
            <LocalValue> taskB_newSoln.d </LocalValue>
        </SetOutIn>
        <SetOutIn port="nIter">
            <LocalValue> taskB_nIter.d </LocalValue>
        </SetOutIn>
        <SetOutput port="oldCost">
            <LocalValue> taskB_oldCost.d </LocalValue>
        </SetOutput>
    </DataMux>
    <RepeatCondition>
        <UserRepeat>
            evalRepeatCondition \n            "[cdtReadData newCost]" IsLessThan "[cdtReadData oldCost]"
        </UserRepeat>
    </RepeatCondition>
</TaskInstance>
<TaskInstance instance="(C)" maxIterate="15">
    <DataMux>
        <SetInput port="root">
            <LocalValue> taskC </LocalValue>
        </SetInput>
</SetInput>
<SetInput port="inpDescr">
    <LocalValue> taskC_inpDescr.d </LocalValue>
</SetInput>
<SetInput port="initCost">
    <LocalValue> 1.e6 </LocalValue>
</SetInput>
<SetInput port="initSoln">
    <LocalValue> 23 </LocalValue>
</SetInput>
<SetOutIn port="newCost">
    <LocalValue> taskC_newCost.d </LocalValue>
</SetOutIn>
<SetOutIn port="newSoln">
    <LocalValue> taskC_newSoln.d </LocalValue>
</SetOutIn>
<SetOutput port="nIter">
    <LocalValue> taskC_nIter.d </LocalValue>
</SetOutput>
<SetOutput port="oldCost">
    <LocalValue> taskC_oldCost.d </LocalValue>
</SetOutput>
</DataMux>
<RepeatCondition>
    <UserRepeat>
        evalRepeatCondition \ 
        "[cdtReadData newCost]" IsLessThan "[cdtReadData oldCost]"
    </UserRepeat>
</RepeatCondition>
</TaskInstance>
<TaskInstance instance="(D)">
    <JoinCondition>
        <UserInvoke>
            evalJoinCondition OR \ 
            [evalJoinCondition AND {A} Valid {B} Valid {C} Valid] \ 
            [evalJoinCondition AND {A} Valid {B} Valid {C} Skipped] \ 
            [evalJoinCondition AND {A} Valid {B} Skipped {C} Valid] \ 
            [evalJoinCondition AND {A} Valid {B} Skipped {C} Skipped] \ 
            [evalJoinCondition AND {A} Skipped {B} Valid {C} Valid] \ 
            [evalJoinCondition AND {A} Skipped {B} Valid {C} Skipped] \ 
            [evalJoinCondition AND {A} Skipped {B} Skipped {C} Valid] \ 
            [evalJoinCondition AND {A} Skipped {B} Skipped {C} Skipped] \ 
        ;
    </UserInvoke>
</JoinCondition>
<DataMux>
    <SetInput port="costA">
        <LocalValue> taskD_costA.d </LocalValue>
    </SetInput>
    <SetInput port="costB">
        <LocalValue> taskD_costB.d </LocalValue>
    </SetInput>
</DataMux>
<SetInput>
<SetInput port="costC">
   <LocalValue> taskD_costC.d </LocalValue>
</SetInput>
<SetOutput port="totalCost">
   <LocalValue> taskD_totalCost.d </LocalValue>
</SetOutput>
<SetOutput port="costAvg">
   <LocalValue> taskD_costAvg.d </LocalValue>
</SetOutput>
<SetOutput port="count">
   <LocalValue> taskD_count.d </LocalValue>
</SetOutput>
</DataMux>
</TaskInstance>
<TaskInstance instance="(E)"/>
<DataMux>
<SetInput port="totalCost">
   <LocalValue> taskE_totalCost.d </LocalValue>
</SetInput>
<SetInput port="costAvg">
   <LocalValue> taskE_costAvg.d </LocalValue>
</SetInput>
<SetInput port="count">
   <LocalValue> taskE_count.d </LocalValue>
</SetInput>
<SetInput port="outReport">
   <LocalValue> taskE_outReport.d </LocalValue>
</SetInput>
</SetOutput>
</DataMux>
</TaskInstance>
</EndJoin/>
<!-- DataGraph for multi-task body "task_parabola" -->
<DataGraph>
   emailaddr -> (InitA).emailaddr
   emailaddr -> (InitB).emailaddr
   emailaddr -> (InitC).emailaddr
   mInDescrA -> (InitA).inpDescr
   mInDescrB -> (InitB).inpDescr
   mInDescrC -> (InitC).inpDescr
   (InitA).emailaddr -> (A).emailaddr
   (InitB).emailaddr -> (B).emailaddr
   (InitC).emailaddr -> (C).emailaddr
   mInDescrA -> (A).inpDescr
   mInDescrB -> (B).inpDescr
   mInDescrC -> (C).inpDescr
   mInitCost -> (A).initCost
   mInitCost -> (B).initCost
   mInitCost -> (C).initCost
   (A).newCost -> (D).costA
(B).newCost  ->  (D).costB
(C).newCost  ->  (D).costC
(D).totalCost ->  (E).totalCost
(D).costAvg  ->  (E).costAvg
(D).count   ->  (E).count
(E).outReport ->  mOutReport
</DataGraph>
</MultiTaskBody>

<SingleTaskBody name="pInitSolver" sleep="1">
  <ExecCommand type="TclScript">
    <Value>
      # First, delete all files with root and extension '.d'
      foreach f [glob -nocomplain *[cdtGetData root]*.d] {
        file delete -force $f
      } ;# End of foreach loop
      file delete -force [cdtGetData inpDescr]
      # Now, copy data file from repository
      foreach f [glob -nocomplain [file join [cdtGetData initDataDir] [cdtGetData root]*.d]] {
        file copy $f .
      } ;# End of foreach loop
      set f [file join [cdtGetData initDataDir] [cdtGetData inpDescr]]
      if {![file exists $f]} {
        set f [file join [cdtGetData initDataDir] [cdtGetData root]_inpDescr.d]
      } ;# End of if stmt
      file copy $f [cdtGetData inpDescr]
    </Value>
  </ExecCommand>
</SingleTaskBody>

<MultiTaskBody name="pSolverFlow">
  <BeginFork/>
  <TaskInstance instance="(pSolverWeb)">
    <DataMux>
      <SetInput port="inpDescr">
        <LocalValue> [cdtGetData root]_inpDescr.d </LocalValue>
      </SetInput>
      <SetOutIn port="newCost">
        <LocalValue> [cdtGetData root]_newCost.d </LocalValue>
      </SetOutIn>
      <SetOutIn port="newSoln">
        <LocalValue> [cdtGetData root]_newSoln.d </LocalValue>
      </SetOutIn>
      <SetOutput port="nIter">
        <LocalValue> [cdtGetData root]_nIter.d </LocalValue>
      </SetOutput>
      <SetOutput port="oldCost">
        <LocalValue> [cdtGetData root]_oldCost.d </LocalValue>
      </SetOutput>
    </DataMux>
  </TaskInstance>
</MultiTaskBody>
<SetOutput port="results">
  <LocalValue> [cdtGetData root].results.web </LocalValue>
</SetOutput>
</DataMux>
</TaskInstance>
<TaskInstance instance="(download)">
  <DataMux>
    <SetOutput port="newCostLcl">
      <LocalValue> [cdtGetData newCostWeb] </LocalValue>
    </SetOutput>
    <SetOutput port="newSolnLcl">
      <LocalValue> [cdtGetData newSolnWeb] </LocalValue>
    </SetOutput>
    <SetOutput port="nIterLcl">
      <LocalValue> [cdtGetData nIterWeb] </LocalValue>
    </SetOutput>
    <SetOutput port="oldCostLcl">
      <LocalValue> [cdtGetData oldCostWeb] </LocalValue>
    </SetOutput>
  </DataMux>
</TaskInstance>
</EndJoin/>

!-- DataGraph for multi-task body "pSolverFlow" -->
<DataGraph>
  emailaddr   -> (pSolverWeb).emailaddr
  root        -> (pSolverWeb).root
  inpDescr    -> (pSolverWeb).inpDescr
  initCost    -> (pSolverWeb).initCost
  initSoln    -> (pSolverWeb).initSoln

  emailaddr   -> (download).emailaddr
  (pSolverWeb).newCost  -> (download).newCostWeb
  (pSolverWeb).newSoln -> (download).newSolnWeb
  (pSolverWeb).nIter   -> (download).nIterWeb
  (pSolverWeb).oldCost -> (download).oldCostWeb

  (download).newCostLcl  -> newCost
  (download).newSolnLcl  -> newSoln
  (download).nIterLcl    -> nIter
  (download).oldCostLcl  -> oldCost
</DataGraph>
</MultiTaskBody>

<SingleTaskBody name="pSolver" sleep="1">
  <ExecCommand type="HttpCmd">
    <Value>
      cdtHttpSubmit "[cdtGetData parabolurl]" \\
      "EmailAddr  [cdtGetData emailaddr] \\
      inpDescr   [cdtGetData inpDescr] \\
      initCost   [cdtGetData initCost] \\
      (download).newCostWeb \\
      (download).newSolnWeb \\
      (download).nIterWeb \\
      (download).oldCostWeb \\
      newCost \\
      newSoln \\
      nIter \\
      oldCost
    </Value>
  </ExecCommand>
</SingleTaskBody>
initSoln [cdtGetData initSoln] \ 
newCost [cdtGetData newCost] \ 
newSoln [cdtGetData newSoln] \ 
oldCost [cdtGetData oldCost] \ 
nIter [cdtGetData nIter]" \ 
"[cdtGetData results]"

</Value>
</ExecCommand>
</SingleTaskBody>

<SingleTaskBody name="download" sleep="1">
<ExecCommand type="HttpCmd">
<Value>
    cdtHttpGet [cdtGetData resultsurl]/[cdtGetData oldCostWeb tail]
    cdtHttpGet [cdtGetData resultsurl]/[cdtGetData newCostWeb tail]
    cdtHttpGet [cdtGetData resultsurl]/[cdtGetData newSolnWeb tail]
    cdtHttpGet [cdtGetData resultsurl]/[cdtGetData nIterWeb tail]
</Value>
</ExecCommand>
</SingleTaskBody>

<SingleTaskBody name="pEvaluator" sleep="1">
<ExecCommand type="TclCmd">
<Value>
    pEvaluator.tcl "[cdtGetData costA]" "[cdtGetData costB]" \ 
    "[cdtGetData costC]" "[cdtGetData totalCost]" \ 
    "[cdtGetData costAvg]" "[cdtGetData count]"
</Value>
</ExecCommand>
</SingleTaskBody>

<SingleTaskBody name="pReport" sleep="1">
<ExecCommand type="TclCmd">
<Value>
    pReport.tcl "[cdtGetData totalCost]" "[cdtGetData costAvg]" \ 
    "[cdtGetData count]" "[cdtGetData outReport]"
</Value>
</ExecCommand>
</SingleTaskBody>
</Cdtml>
<!-- parabola_body.cdt -->