Executable Workflows:

A Paradigm for Collaborative Design on the Internet

Hemang Lavana Amit Khetawat Franc Brglez Krzysztof Kozminski

CBL (Collaborative Benchmarking Laboratory)
Department of Computer Science
Box 7550, NC State University,
Raleigh, NC 27695, USA

National Semiconductor Corp.
2900 Semiconductor Drive,
P.O. Box 58090 M/S D3-677,
Santa Clara, CA 95052-8090

http://www.cbl.ncsu.edu/demos

Abstract – *This paper introduces a directed hypergraph model that supports (1) workflow composition and reconfiguration while accessing and executing programs, data, and computing resources across the Internet, (2) synchronous and asynchronous peer-to-peer interaction between members of any team during workflow composition and execution, (3) synchronous and asynchronous peer-to-workflow interaction between any team member and any object in the workflow.*

Given a library of program and data nodes, editing the workflow and its execution is as intuitive as the hierarchical schematic design capture and simulation. Examples of multi-site multi-user applications demonstrate that the proposed workflow implementation provides a user-friendly paradigm for distributed and collaborative team design.

I. Introduction

Considerable research is being directed in support of collaborative activities using the Internet. For example, the paper [1] provides good insights on issues related to sharing applications. Systems that are currently available include Xplexer [2], Xshare[3], XMX [4], XTV [5], and ShowMe SharedApp[6]. However, none of these contribute directly to the collaborative Internet-based workflow paradigm described in this paper.

We consider the workflow simply as an *executable directed hypergraph*, with nodes representing programs and data, and hyperedges representing data-to-program, program-to-data, data-to-data, and program-to-program dependencies. Both data and programs can reside on any host with a unique IP address. Program nodes can be hierarchical: they may expand into their own workflows of data and program nodes. Workflow transformations to Petri nets [7] give rise to (1) a canonical, and (2) an executable Petri net representation; the latter generates the schedule for workflow execution.

User teams, *at different sites*, control the workflow execution by selection of its paths. When executed, *all* users can observe blinking edges for any data/code transfers between nodes on different hosts, and blinking nodes for any programs executing on a particular host (programs on different hosts can execute in parallel). By highlighting any data node, the user can select/view/edit from a displayed list of files associated with the particular data node, be it data that is read by the program node, or data generated by the program node. Reconfiguring workflows is as simple as editing a netlist schematic. Program node and data node template editors are provided to encapsulate all programs and data, such that semantic consistency of the workflow can be maintained.

We use `Tcl/Tk` and `Expect` [8, 9, 10] to implement a GUI and a collaborative team environment, originally initiated as a `RE`configurable and re`U`sable `B`enchmarking `EN`vironment, `REUBEN`. As illustrated in the paper, collaborative benchmarking applications may indeed be the first realistic test of this environment. Programs that have been encapsulated to date on local and remote hosts, currently without relying on CORBA [11], include university–based public–domain tools, commercial tools, and a number of Web browsers invoking JAVA applications anywhere on the Internet.

The approach we propose and have implemented relates to earlier work in the area of design methodologies and design frameworks such as [12, 13, 14, 15, 16]. Complementary workflow definitions and applications cover a wide range, e.g. they arise in the context of office automation [17], database and scientific applications [18, 19], and schemas [20].

This paper is organized into the following sections: (2) an introductory example of an executable workflow; (3) workflow construction and its transformations, deriving the schedule for workflow execution; (4) user view of the workflow; (5) mechanisms to support a collaborative workflow environment; (6) examples of several multi-site multi-user applications.

II. Motivation

We use a simple example to contrast a traditional workflow approach with the one proposed in this paper.

Manual Workflow Example. Consider the three sites depicted in Figure 1. Site 1 (in CA) specializes in FPGA placement and layout of netlists that fit into technology-specific library devices. Site 2 (in MD) specializes in partitioning large netlists, given the constraints of a technology-specific device library. Site 3 (in TX) specializes in logic optimization, given a reasonably sized netlist. Assume that the team at Site 1 synthesized a design that exceeds the capacity of the largest device, and recognized the need for a `netlist partitioner` such as the one at Site 2, and a `logic optimizer` such as one at Site 3. The team at Site 1 has the following choices:

1. Download and install(!) `partitioner` from Site 2 and `optimizer` from Site 3.
2. Get accounts on Sites 2 and 3 and rely on `telnet` and `ftp` to move data and access tools at each site. Figure 1 shows a flow of typical steps performed manually.
3. Get accounts on Sites 2 and 3 and create an *executable workflow* such as shown in Figure 2 that provides a consistent and efficient implementation of a combined recursive partitioning and optimization paradigm.

This research was supported by contracts from the Semiconductor Research Corporation (94–DJ–553), SEMATECH (94–DJ–800), and DARPA/ARO (P-3316–EL/DAAH04–94–G–2080). During this research, K. Kozminski was with Relative Software, Inc.

DAC 97, Anaheim, California

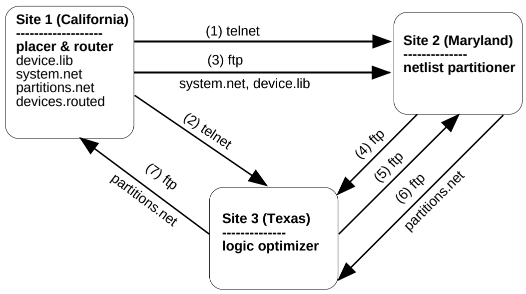


Fig. 1. Steps in a manually executed multi-site workflow.

Whereas the first two choices are time-consuming and error-prone, we argue the third choice to be more effective in terms of the proposed collaborative environment. Downloading a complex tool may be simple, installation and maintenance is not. Even if the installation of all tools is successful, we still don't have a workflow such as one in Figure 2. In the context of the Web, where we maintain lists of URLs rather than big documents or applets, the workflow implemented in Figure 2 has the advantage of tool updates as well as on-site expertise.

Once a site supports an executable workflow environment and broadcasts it simultaneously to the displays of users at participating sites, all sites become aware of the global design objectives, tool and data dependencies to achieve them, and the expertise distributed between the sites. A collaborative and synergistic control mechanism, whereby each site not only observes the action of others but also lends expertise within the relevant part of the workflow (be it partitioning, logic optimization, or placement and routing), has the potential of significantly improving the overall design process when contrasted with improvements achieved in isolation.

Executable Workflow Example. The workflow in Figure 2 is an *executable directed hypergraph*, with nodes representing executable programs and data, while hyperedges represent data-to-program, program-to-data, data-to-data, and program-to-program dependencies. As shown, both data and programs can reside on any host with a unique IP address at any site. While the GUI is designed with color-coded nodes and edges, all nodes in this figure are depicted with white background for improved readability. This is an example of a non-trivial workflow that implements a rather complex partitioning and optimization algorithm, based on the description provided in [21], and we will use it repeatedly to motivate and illustrate the main ideas in our approach.

The algorithm implemented by the workflow in Figure 2 is a recursive application of a bi-partitioning tool that partitions a large netlist into a tentative partition and a remainder. The tentative partition is optimized by a logic optimizer tool, and evaluated for fit into the largest device from the device library. If the optimized partition is not acceptable, variables for the bi-partitioner are adjusted and the bi-partitioner re-executes to generate a smaller tentative partition. Eventually, the optimized partition is acceptable and the partition is placed and routed into a library-based device. The process now repeats on the partition remainder until the complete netlist is partitioned into a number of library devices.

Figure 2 shows that nodes in this workflow have been assigned to Hosts 1, 2, and 3 corresponding to three sites in Figure 1. Nodes on Host 1 consist of data directories that contain the original netlists and the device library as inputs, while the outputs can be viewed as data directories that archive partitions in terms of placed and routed technology-specific devices. `Partitioner` on Host 2 implements the partitioning algorithm, `Optimizer` on Host 3 depicts a hierarchical node

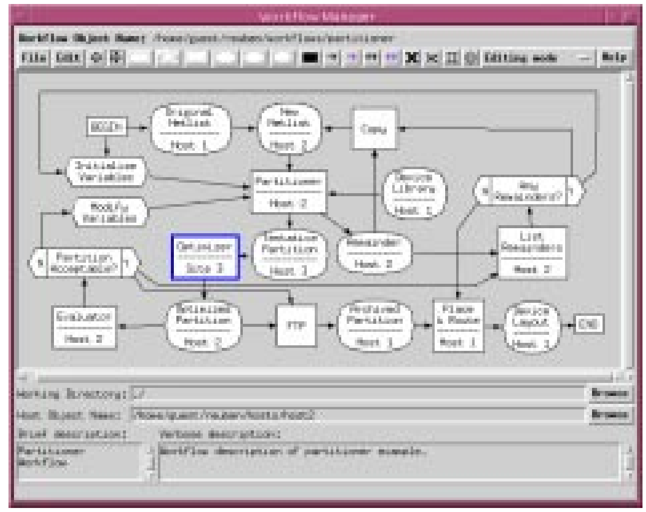


Fig. 2. Example of an Internet-based executable workflow.

that performs logic optimization, `Place & Route` on Host 1 performs placement and routing within the designated FPGA device and is a commercial tool [22]. Data nodes have oval shapes and represent a file directory that contains any number of files of class *data*. Program nodes represent encapsulated programs of arbitrary complexity, interfaced to data and script nodes, and controlled by decision nodes, Unix utilities such as `ls`, `cp`, `ftp`, `telnet`, etc. All relevant information pertaining to hosts, such as IP address and user login name, are defined in a host template. Each node refers to such a host template by a mnemonic host ID as shown in Figure 2. Nodes without a host ID inherit the host ID of the workflow.

Compiler supports simple syntax and semantics constraints. For example, *data-to-program* dependencies must satisfy a range of pre-configured matching pattern constraints: only data files that match the program template pattern are accepted when creating dependency edges to a given program. Semantically, an unconditional *data-to-data dependency*, in the context of a data node on host i and a data node on host j, implies an unconditional and *explicit file transfer* of a data file from host i to host j. In this case, data nodes at host i and host j are shown explicitly. On the other hand, an unconditional *data-to-program* dependency of a data node on host i and a program node on host j implies an unconditional and *implicit file transfer* of a data file from host i to host j. In this case, the data node at host j is not shown explicitly. We have examples where both concepts are useful. In addition to satisfying simple workflow-specific syntax and semantics constraints, the workflow is also expected to *execute correctly under all valid input data*. For example, a data node can be written and over-written by several program nodes in the workflow, but *never* at the same point in time.

III. Multi-Site Executable Workflow: Formal View

A user-generated workflow is a concise description of several heterogeneous applications in a directed hypergraph. A well-defined procedure is required to guarantee successful execution of a workflow, such as one in Figure 2, given that it may be concurrent, asynchronous, and distributed. We find it convenient to use Petri nets as a modeling tool for describing and executing such a workflow. A *Petri net* [7] is a directed, weighted, bipartite graph consisting of two kinds of nodes: *places* and *transitions*, and edges connecting nodes of different kinds.

A user-generated workflow is systematically transformed

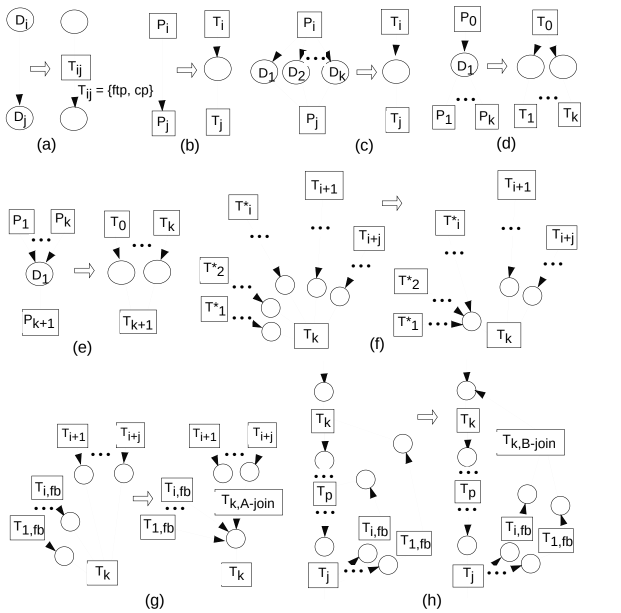


Fig. 3. Workflow to executable Petri net transformations.

into an executable Petri net in five steps: (1) Generate a canonical Petri net representation; (2) Identify the cycles and feedback nodes; (3) Transform the canonical representation into an executable Petri net; (4) Generate a firing schedule for all transitions; (5) Execute the workflow.

Step1: Canonical Petri net representation. Six rules *R1-R6* transform a workflow into a canonical Petri net representation. The basis for the rules is a one-to-one correspondence of program nodes P_i in the workflow and the transitions T_i in the Petri net. It is important to distinguish between a regular program node P_i and a decision node P_i^*; we have T_i and T_i^* transition nodes in a Petri net. The correspondence of data nodes D_i to a place in the Petri net is subject to the transformations.

R1 *Every program node P_i is replaced by a transition T_i; every decision node P_i^* is replaced by a transition T_i^*; and every data node D_i is replaced by a place.*

R2 *Whenever a data node D_i drives a data node D_j, the edge connecting the two is replaced with two edges and a new transition T_{ij} between two places, as shown in Figure 3(a).* This new transition T_{ij} may signify nodes such as *copy, ftp, ..., etc*, depending on whether the two places represent data nodes on the same or different hosts. In Figure 2, an edge from `Original_Netlist` on *host 1* to `New_Netlist` on *host 2* implies `FTP` of data.

R3 *Whenever a program node P_i drives another program node P_j, a conceptual place is introduced between their respective transitions T_i and T_j, as shown in Figure 3(b).* Such a place indicates completion of the transition T_i. Figure 2 depicts such a program node `Evaluator` driving the `Partition_Acceptable?`.

R4 *If a program node P_i drives data nodes $D_1, D_2, \ldots, D_k$ and the same data nodes drive a program node P_j, then a single place replaces all data nodes between transitions T_i and T_j, as shown in Figure 3(c).*

R5 *Whenever a data node is driven by program node P_0 and drives program nodes $P_1, P_2, \ldots, P_k$, we have k edges from transition T_0 driving k places before reaching transitions $T_1, T_2, \ldots, T_k$, as shown in Figure 3(d).* In Figure 2, data node `Optimized_Partition` is driven by `Optimizer` and drives `Evaluator` and `FTP`.

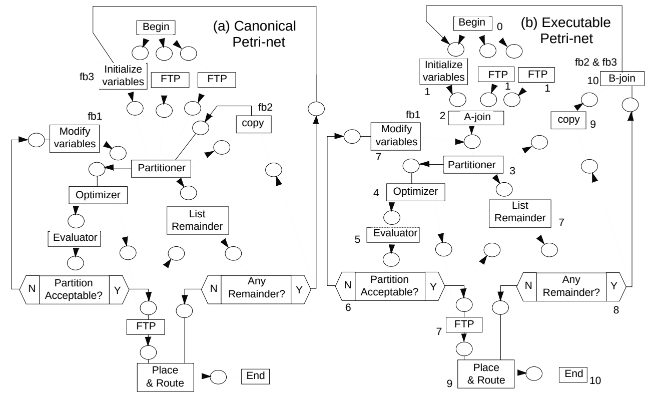


Fig. 4. Canonical and executable Petri net representations.

R6 *Whenever a data node is driven by program nodes $P_1, P_2, \ldots, P_k$ and drives a program node P_{k+1}, we have k edges from transitions $T_1, T_2, \ldots, T_k$ driving k places before reaching transition T_{k+1}, as shown in Figure 3(e).* In Figure 2, data node `New_Netlist` drives `partitioner` node and is driven by a `Copy` and an implied `FTP` nodes.

Rule *R1* must be applied to *all* workflows to transform any workflow into its canonical representation. Depending on the structure of the workflow, one or more of rules *R2-R6* may be required to complete the transformation. The canonical Petri net is a unique representation of the underlying workflow. If acyclic, the canonical representation itself is executable, since the firing schedule can be readily derived.

Step2: Cycles in a workflow. Cycles typically occur when certain programs need to be executed repeatedly. In every cycle, at least one node can be identified as a feedback node, signifying a return to the top of the loop in the workflow.

The transition at the top of the loop can never be enabled since tokens cannot be generated for input places which are driven by feedback paths. Hence, canonical Petri nets that are cyclic are not executable. This problem is overcome by identifying, in a canonical Petri net, all cycles and their feedback nodes that exist in the graph, and applying rules *R7-R9* to create an executable Petri net model. In Figure 4(a), three feedback nodes have been identified: two driving the `Partitioner` and one driving the `Initialize_Variables`.

Step3: Executable Petri Net Representation. Transformations *R7-R9* create an executable Petri net.

R7 *All places that are on paths of transitions $T_1^*, T_2^*, \ldots, T_i^*$, and incident at transition T_k are replaced by a single place incident at transition T_k, as shown in Figure 3(f).*

R8 *All places incident at transition T_k and driven by transitions $T_{1,fb}, T_{2,fb}, \ldots, T_{i,fb}$, in the feedback vertex set are replaced by a single place incident at transition T_k. In addition, the single place incident at transition T_k is also driven by a new transition $T_{k,A-join}$ which is introduced to merge all remaining places incident at transition T_k, as shown in Figure 3(g).* In Figure 4(a) and (b), transitions `Modify_Variables`, `Copy`, `Partitioner`, `FTP` and `Initialize_Variables` depict such a situation and its corresponding transformation. Note this rule is not applied to the feedback transition `Copy` since *R9* is applicable.

R9 *All places that are on path of transition T_j and driven by transitions $T_{1,fb}, T_{2,fb}, \ldots, T_{i,fb}$, in the feedback vertex set are merged to drive a new transition $T_{k,B-join}$. The new transition $T_{k,B-join}$ drives the place incident at transition T_k, as shown in Figure 3(h).* In Figure 4(a), <Any_Remainder?→Copy→Partitioner> and <Any_Remainder?→Initialize_Variables> are two such feedback paths, their corresponding transformations are shown in Figure 4(b).

Host1											
Begin	⋆										
End											⋆
Place and Route										⋆	
FTP		⋆									
FTP		⋆									
Host2											
A_join			⋆								
B_join											⋆
Initialize Variables		⋆									
Modify Variables								⋆			
FTP								⋆			
Copy										⋆	
List Remainder								⋆			
Partitioner				⋆							
Evaluator						⋆					
Partition Acceptable?							⋆				
Any Remainder									⋆		
Host3											
Optimizer					⋆						
Level of Petri net transition	0	1	2	3	4	5	6	7	8	9	10

Fig. 5. Firing schedule of an executable Petri net.

Step4: A firing schedule. Given an executable Petri net, the firing schedule for all the transitions is easily generated, as shown in Figure 5. Each transition has a level at which it may be fired, indicated by a ⋆. Two or more transitions occurring at the same level: (1) fire concurrently if on different hosts, (2) fire sequentially if on the same host.

Step5: Workflow execution. We have evolved a simple yet sufficiently powerful methodology to abstract any user-generated workflow into an executable workflow by use of Petri nets. We further realize that a one-to-one correspondence exists between program nodes and transition nodes. Hence, every transition node in a Petri net can be thought of as a virtual node which always fires and generates tokens when enabled, irrespective of whether its corresponding program node requires execution or not.

Whenever a transition node is enabled, its corresponding program node is examined for its input and output data node dependencies. A program node re-executes only if any of its output data is missing or its time-stamp is older with respect to any of its input data or programs. Depending on the type of program-data dependency, such as *main*, *optional*, *multiple* or *matching*, a re-executable program node invokes as many times as there exist data files of *main* dependency, but only once for every *multiple* type of dependency. Report [23] describes more details. Figure 6 shows the `Partitioner` program node with its input and output data file dependencies.

For multi-site nodes, where the program and data nodes reside on several hosts, there may be a non-trivial time difference between the GMT times of the two hosts. Hence, time-stamps of such data files are adjusted by δ (on the order of a few minutes) before examining such nodes for re-execution. Program re-execution, if required, results in data file transfers among hosts before and after its invocation. Thus, while a program node may not execute at all, the corresponding transition node will still generate its output token(s).

Having formalized an effective way of modeling workflows, we next highlight some of the aspects of creating and editing workflows. Complete details are available in the user manual.

IV. Multi-Site Executable Workflow: Editing View

The graphical user interface of the workflow consists of a number of specialized windows.

Workflow Editor. A casual user may only be interested in editing the workflow using an existing library of predefined `REUBEN` objects such as program nodes, data nodes, decision boxes, hierarchical workflow nodes, etc. These objects

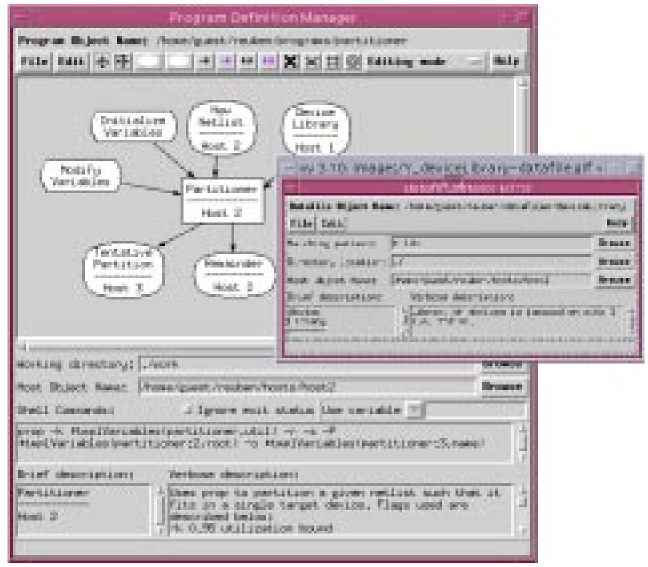


Fig. 6. Program and Data Node Editor windows.

are instantiated in the workflow in a manner similar to the schematic capture of a gate from a device library.

Standard features present in many drawing editors, such as adjustable grid spacing, undo/redo, etc., are provided to aid in workflow composition. Editing operations include adding, deleting, and rearranging nodes, as well as hooking them together with links representing the dependencies.

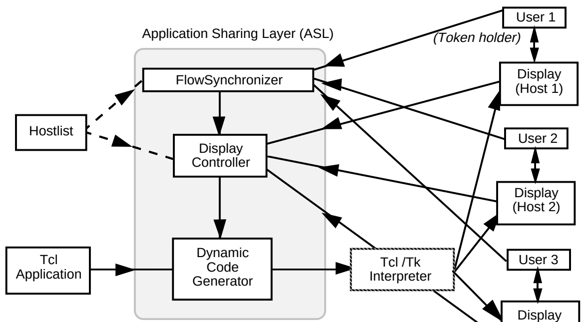
Program and Data Node Editor. Figure 6 shows the elements used in defining a program node `Partitioner` on `Host 2` and a data definition editor for `Device Library` on `Host 1`. Input and output data dependencies in relation to the program are specified by creating a link of appropriate type between the two in the graphical window. The lower portion of the editor consists of various fields used to specify commands to execute on program invocation, the working directory and the host on which to invoke the program node. Other fields, like variables, allow the user to initialize and dynamically modify, using script nodes, parameters passed to the command arguments during execution. Important fields for defining a data node are: (1) data filename matching pattern, (2) directory, and (3) host location.

V. Multi-Site Executable Workflow: Collaborative View

Designers working on complex projects may be distributed in space and time. Current methods to support distributed team collaboration include e-mail, phone, and relatively expensive video conferencing between remote sites.

The Internet, intranets and powerful desktop workstations offer an opportunity to dramatically improve real-time collaborative environments. Team members can interact synchronously with applications on displays attached to their hosts. All members are given the capability to control, one at a time, an application of common interest, and all members have the opportunity to observe the results.

We considered three methods to implement a collaborative environment for Tcl/Tk applications: (1) *Tcl/Tk architecture modification* by incorporating an intermediate layer between the Tcl/Tk application code and the Tcl/Tk Interpreter such that it dynamically generates code to draw graphics on multiple displays as well as receive events from them; (2) *Multiple interpreter invocation* for each user in the session to control the events, such as mouse clicks and drawings, on each display



Original Application (Hello):


```
label .hello -text "Hello World"
pack .hello
```

Multi-cast Application (M-Hello) generated by ASL:

```
DisplayController and FlowSynchronizer initialization:
  toplevel .screen1 -screen screen@site1:0
  toplevel .screen2 -screen screen@site2:0
  toplevel .screen3 -screen screen@site3:0
Dynamic code generation:
  label .screen1.hello -text "Hello World"; pack .screen1.hello;
  label .screen2.hello -text "Hello World"; pack .screen2.hello;
  label .screen3.hello -text "Hello World"; pack .screen3.hello;
```

Fig. 7. Multi-user *collaborative* architecture.

and, in turn, communicate with a master interpreter which manages the application; (3) *Double Buffering* in two phases, first routing all graphics data to an off-screen buffer, and then updating the changes in this buffer to multiple displays. We choose to implement the first method since double buffering requires large bandwidth and multiple interpreter is difficult to implement.

Multi-user collaborative environment. The top portion of Figure 7 shows an *Application Sharing Layer* (ASL) between the Tcl/Tk application code and the Tcl/Tk Interpreter. This layer consists of three components: *FlowSynchronizer* to facilitate real-time conferencing, *Display Controller* to manage control exchanges among the collaborators and *Dynamic Code Generator (DCG)* to multicast the application as well as process incoming events from the collaborator's displays. The user designated as *Token Holder* can control the application.

A simple application `Hello` to illustrate the essential ideas of the multicasting operation is shown in the bottom portion of Figure 7. Original application code and its corresponding display window are adjacent in the figure, followed by code for multicast application `M-Hello` shown in two parts:

- *Display Controller and FlowSynchronizer initialization.* Initialization process uses `toplevel` command to invoke the FlowSynchronizer window on each display. It also sets up a control manager to furnish the designated *Token Holder* with the initial control of the application.
- *Dynamic Code Generation.* This intercepts all `Tcl/Tk` commands and dynamically generates multiple commands, one for each display, before passing it to the interpreter.

Control in a collaborative multi-user environment. In most applications, it is essential that at any time during the execution of the application, only one user controls the input. Several methods proposed for allocation of control to multiple users include:

1. *No Control.* All users can execute or terminate the application at any time, resulting in contentions.
2. *Single User Control.* A single user controls the entire application, while others are mere spectators.
3. *Automated Control.* The control is pre-configured to switch between the end-users based on their actions, such as in a chess game, both players secure control alternatively after each move.
4. *Floating Control.* The application is invoked without any user-control; the first user to grab it get the controls.
5. *Request-based Control.* The initial control is specified at the time the application is launched; however, another user may acquire control upon request later on.

We use a request-based control mechanism to exchange control among collaborators. It is not necessary to use C or C++ in order to manipulate the widgets, and useful applications can be built very rapidly using Tcl/Tk. This is in sharp contrast to groupware tools such as Xplexer and Xshare, which use the Xlib and Motif toolkits for implementation.

A Display of a Collaborative Workflow. Upon the initial invocation of a collaborative workflow with `REUBEN`, users at *all* sites typically see two windows, such as shown in Figure 8: an application specific workflow, and the `FlowSynchronizer`. We say more about the purpose of this workflow in the next section. The purpose of `FlowSynchronizer` is two-fold: (1) to transmit typed messages between team members, and (2) to dynamically exchange control using a request-based mechanism. The `FlowSynchronizer` can support n collaborating sites, providing controlled access to two window segments at each site: (a) a clickable button designating the *UserSite*, and (b) a scrollable window which provides a real-time conferencing environment. At any time, one and only one site is designated as a *TokenHolder*, by coloring its *UserSite* button in a color different from all other sites. At any time, each collaborator can transmit text messages in the scrollable window to all other sites. However, only the *TokenHolder* has the capability to click on another *UserSite* button to pass the token, and hence the control of the entire environment, including any application and data displayed in the workflow window.

Additional details about the implementation, and experience with the collaborative processes, are given in [23].

VI. Multi-Site Executable Workflow: Experiments

Our web site (`http://cbl.ncsu.edu/demos/`) supports `REUBEN` demos that include several collaborative workflow experiments. In this paper, we briefly describe only two.

A Collaborative Experiment with WELD. The collaborative workflow example in Figure 8 has been tested with members of the WELD team at UC Berkeley in Sept. 1996 (`..//www-cad.EECS.Berkeley.EDU/Respep/Research/weld/`). The main purpose of the workflow is to demonstrate a multi-site workflow that encapsulates tools such as the FSM editor accessible as browser node through its URL (a JAVA-applet `http://yoyodyne.EECS.berkeley.EDU/fsm/fsm.html`), a university-based tool SIS (synthesis and logic optimization tool, [24]), and a commercial tool APR (automatic placement and routing tool [22]). While FSM editor is executed at the UCB site, both SIS and APR are executed at CBL site. As shown in the `FlowSynchronizer` window, the workflow is launched from zodiac@CBL and is multicast to yoyodyne@berkeley. The user at CBL is passing control to a specialist at the UCB site for assistance with the FSM editor. At some other time, a user at CBL may choose to enter the FSM design specifications on his/her own. Once the control is passed back to CBL, the design specifications are transferred to CBL, optimized by the SIS tool and mapped to FPGAs using Xilinx's automatic place and router (APR). CBL may pass control to UCB so the APR's layout can be reviewed at the UCB site. CBL may be requested to manually route the netlist, or there may be a change of FSM specifications that will result in a new layout with improved timing performance.

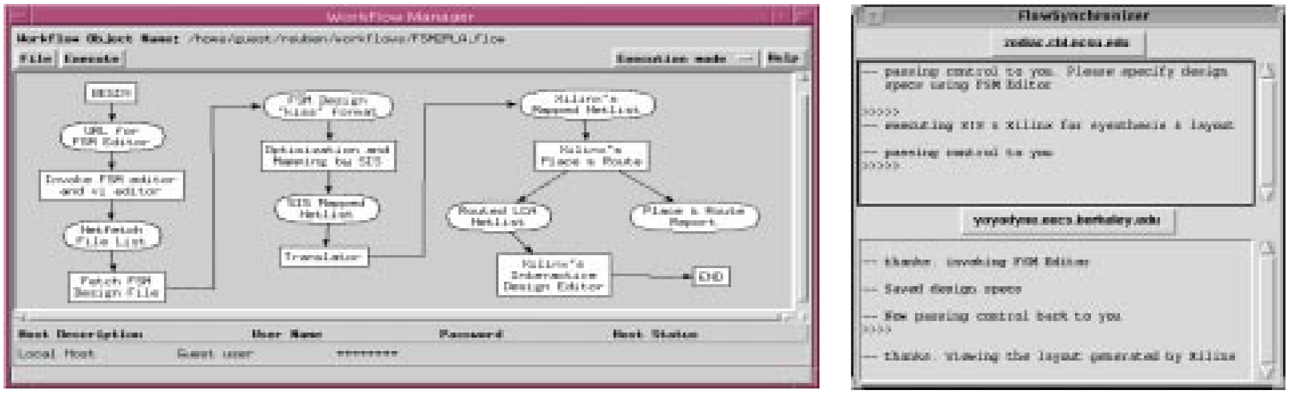


Fig. 8. Typical user views in a multi-user collaborative environment.

Collaborative Benchmarking Experiment. The workflow in Figure 9 illustrates a process we have tested to perform a series of collaborative Internet-based benchmarking experiments which in this case may involve: (1) two technology mappers using the same library, (2) two libraries using the same technology mapper, or (3) a combination of both. As described earlier, a window named `TechnologyMapping` is displayed at multiple sites, while the window named `FlowSynchronizer` which coordinates token passing between all participants. There may be up to three teams participating in this experiment, each at a site that may be a continent away from the other. We show both data and programs (technology mappers, verification tools) residing at one of the three sites and supported by three hosts. Host 0 archives data (benchmarks) to be used by both technology mappers. Upon completion of technology mapping at either Host 1 or Host 2, results are transferred back to Host 0 for logic equivalence verification, report generation, and archival. As described earlier, the workflow can be executed collaboratively in a synchronous mode (by selecting any of its path segments) or in a batch mode.

We are interested in discussing options with potential participants about testing, hosting, and facilitating benchmarking experiments within the context described in this and the companion paper [25]. For more details, send e-mail to benchmarks@cbl.ncsu.edu with the following information in the body of the message: `subscribe demos`

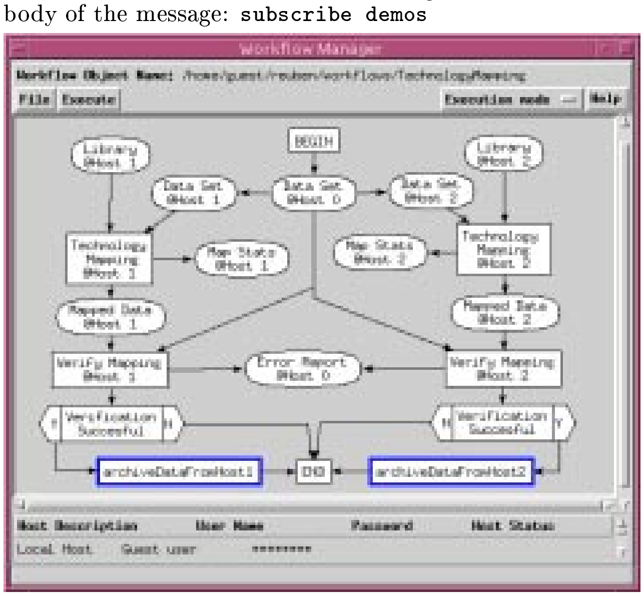


Fig. 9. A collaborative benchmarking experiment.

Acknowledgements. We appreciate the access to tools used in this research: SIS and WELD from teams at UC Berkeley, PROP from Roman Kužnar from U. of Ljubljana (Slovenia), and APR from Xilinx Inc.

References

[1] Hussein Abdel-Wahab and Kevin Jeffay. Issues, Problems and Solutions in Sharing X Clients on Multiple Displays. In *Journal of Internetworking Research and Experience*, pages 01–15, March 1994.

[2] The Application Sharing Technology. Published under URL: `http://andru.unx.com/DD/advisor/docs/jun95/jun95.minenko.shtml` , 1995.

[3] XShare: Workstation Conferencing. Published under URL: `http://www.eit.com/software/xshare/` , 1996.

[4] XMX : A X Protocol Multiplexer. Published under URL: `http://www.cs.brown.edu/software/xmx/` , 1990.

[5] XTV: A User's Guide. Published under URL: `http://www.visc.vt.edu/succeed/xtv.html` , 1993.

[6] ShowMe SharedApp . Published under URL: `http://www.sun.com:80/cgi-bin/show?products-n-solutions/sw/ShowMe/products/ShowMe_SharedApp.html` , 1996.

[7] T. Murata. Petri Nets: Properties, Analysis, and Applications. *Proceedings of IEEE*, pages 541–580, April 1989.

[8] J. K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[9] D. Libes. *Exploring Expect*. O'Reilly and Associates, 1995.

[10] The Tcl/Tk Project At Sun Microsystems Laboratories. URL: `http://www.sunlabs.com:80/research/tcl/` , 1997.

[11] Jon Siegel. CORBA Fundamentals and Programming. J. Wiley, 1996.

[12] D. S. Harrison, R. A. Newton, R. L. Spickelmier, T. J. Barnes. Electronic CAD Frameworks. *Proc. of IEEE*, 78(2):1062–1081, Feb 1990.

[13] J. Daniell and S. W. Director. An Object Oriented Approach to CAD Tool Control Within a Design Framework. *IEEE Transactions on Computer-Aided Design*, 10(6):698–713, June 1991.

[14] A. Casotto and A. Sangiovanni-Vincentelli. Automated Design Management Using Traces. *IEEE Transactions on Computer-Aided Design*, 12(8):1077–1095, August 1993.

[15] E. W. Johnson and J. B. Brockman. Incorporating Design Schedule Management into a Flow Management System. In *32nd Design Automation Conference, ACM/IEEE*, pages 82–87, June 1995.

[16] J. Altmeyer, B. Schurmann and M. Schutze. Generating ECAD Framework Code from Abstract Models. In *32nd Design Automation Conference, ACM/IEEE*, pages 88–93, June 1995.

[17] Layna Fischer. The Workflow Paradigm. Future Strategies Inc., 1995.

[18] Munindar P. Singh. Synthesizing Distributed Constrained Events from Transactional Workflow Specifications. In *Proceedings of the 12th International Conference on Data Engineering*, March 1996.

[19] M. A. Vouk and M. P. Singh. Quality of Service and Scientific Workflows. In The Quality of Numerical Software: Assessment and Enhancements, editor: R. Boisvert, Chapman and Hall, pages 77-89, 1997.

[20] J. B. Brockman and S. W. Director. The Schema-based Approach to Workflow Management. *IEEE Transactions on Computer-Aided Design*, 14(10):1445–1267, October 1995.

[21] R. Kužnar and F. Brglez. PROP: A Recursive Paradigm for Area-Efficient and Performance Oriented Partitioning of Large FPGA Netlists. In *IEEE Intl. Conf. on Computer-Aided Design*, November 1995.

[22] Xilinx. *User Guide and Tutorials*. Xilinx Incorporation, 2100 Logic Drive, San Jose, California, 1991.

[23] H. Lavana, A. Khetawat, and F. Brglez. `REUBEN` User Guide. Technical Report 97-TR-Prep@CBL-03.1, CBL, CS Dept., NCSU, Box 7550, Raleigh, NC 27695, 1997. This report is available as a postscript file via http://www.cbl.ncsu.edu/demos.

[24] SIS – Release 1.1. UC Berkeley Software Distribution, September 1992.

[25] N. Kapur and D. Ghosh and F. Brglez. Towards A New Benchmarking Paradigm in EDA: Analysis of Equivalence Class Mutant Circuit Distributions. In *Intl. Symp. on Physical Design*, 1997.