Executable Workflows: A Paradigm for Collaborative Design on the Internet

Hemang Lavana  Amit Khetawat  Franc Brglez  Krzysztof Kozminski

CBL (Collaborative Benchmarking Laboratory)  National Semiconductor Corp.
Department of Computer Science  2900 Semiconductor Drive,
Box 7550, NC State University,  P.O. Box 38990 M/S D3-677,
Raleigh, NC 27695, USA  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 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.

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. Workflows can be shared and executed anywhere on the Internet.

This research was supported by contracts from the Semiconductor Research Corporation (94-D4453), SEMATECH (94-D4-800), and DARPA/ARO (93-5316-EL/DAOA80-04-4-20810). During this research, K. Kozminski was with Relativ Software, Inc.

Permission to make digital/hard copy of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

DAC 97, Anaheim, California
© 1997 ACM 0-89791-920-3/97/06... skilled at designing a combined recursive partitioning and optimization paradigm.
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 which 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 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 host ID 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
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.

**Step 1: 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 $T_{i,j}$ between two places, as shown in Figure 3(a). This new transition $T_{i,j}$ 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 `OriginalNetlist` on host 1 to `NewNetlist` 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 `Partitioner`. 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 $P_i$ and drives program nodes $P_1, P_2, \ldots, P_k$, we have $k$ edges from transition $T_i$ driving $k$ places before reaching transitions $T_1, T_2, \ldots, T_k$, as shown in Figure 3(d). In Figure 2, data node `OptimizedPartition` is driven by `Optimizer` and drives `Evaluator` and FTP.

**Step 2: 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 `InitializeVariables`.

**Step 3: Executable Petri Net Representation.** Transformations R7-R9 create an executable Petri net.

- **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 $T_k, T_{k+1}, \ldots, T_k$ driving $k$ places before reaching transition $T_k+1$, as shown in Figure 3(e).

In Figure 2, data node `NewNetlist` drives `partitioner` node and is driven by a `Copy` and an implied FTP node. 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.

**Step 4: Execution of workflow.** Figure 4 depicts such a situation and its corresponding transformation. Note this rule is not applied to the feedback transition `Copy` since R9 is applicable.

- **R7** All places that are on paths of transitions $T_1, T_2, \ldots, T_k^*$, 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 $T_{k+1}, T_{k+2}, \ldots, T_n$, 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 $T_{k+1}$, 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 `ModifyVariables`, `Copy`, `Partitioner`, FTP and `InitializeVariables` depict such a situation and its corresponding transformation.

**Step 5: Execution of workflow.** New transitions $T_{k+1}$, in the feedback vertex set are merged to drive a new transition $T_{k+1}$. The new transition $T_{k+1}$ 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?`→`InitializeVariables` are two such feedback paths, their corresponding transformations are shown in Figure 4(b).
Step 1: 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.

Step 2: 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 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.
Several methods proposed for allocation of control to multiple 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 Xplane 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 window.

Additional details about the implementation, and experience with the collaborative processes, are given in [29].

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 section, 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. 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 Xplane and Xshare, which use the Xlib and Motif toolkits for implementation.

**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 window.

Additional details about the implementation, and experience with the collaborative processes, are given in [29].
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 Technology Mapping 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@bl.ncsu.edu with the following information in the body of the message: subscribe demos

![Figure 8](image1.png)

**Figure 8.** Typical user views in a multi-user collaborative environment.

**Figure 9.** A collaborative benchmarking experiment.

**References**


