A Universal Client for Distributed Networked Design and Computing*

Franc Brglez
Dept. of Computer Science
NC State University
Raleigh, NC 27695, USA
brglez@cbl.ncsu.edu

Hemang Lavana†
Cisco Systems, Inc.
7025 Kit Creek Road, P.O. Box 14987
Research Triangle Park, NC 27709, USA
hlavana@cisco.com

ABSTRACT

We introduce a *universal client (OmniFlow)* whose GUI can be readily configured by the user to invoke any number of applications, concurrently or sequentially, anywhere on the network. The design and the implementation of the client is based on the principles of taskflow-oriented programming, whereby we merge concepts from structured programming, hardware description, and mark-up languages. A mark-up language such as XML supports a well-defined schema that captures the decomposition of a program into a hierarchy of tasks, each representing an instance of a blackbox or a whitebox software component. The HDL-like input/output port definitions capture data-task-data dependencies. A highly interactive hierarchical GUI, rendered from the hierarchical taskflow descriptions in extended XML, supports structured programming language constructs to control sequences of task synchronization, execution, repetition, and abort.

Experimental evaluations of the prototype, up to 9150 tasks and the longest path of 1600 tasks, demonstrate the scalability of the environment and the overall effectiveness of the proposed architecture for a number of networked design and computing projects.

1. INTRODUCTION

The Internet is not only changing ways of how designers access tools and data but also how designers organize their projects, given the environments in which they execute them. A *ToolWire* client [1], executable in a web-browser, is an example of a commercial service, pacing the user to click through a sequence of tasks, using icons such as *upload a VHDL file*, *analyze file*, *synthesize an FPGA device*, *generate a report*. The user has no opportunity to expand the repertoire of available tools or to change the sequence of tasks performed by the tools. University-based environments such as *Reuben* [2], *JavaCADD* [3], *WELD* [4], *CollabTop* [5], *OpenDesign* [6], are more diversified. With the *JavaCADD* client, user can access a number of tools in the sequence of their choice, however no sequence is enforced explicitly by the interface. The *CollabTop* client allows two or more users to edit a schematic and invoke a simulator. The *Reuben* client represents the first generation of a *user-configurable* GUI environment to create executable workflows of tools and data on the network, an approach that has been formalized and extended to asynchronous and synchronous collaborative OmniFlow/OmniDesk environments in [7]. The OpenDesign is a prototype environment created with the first generation of the OmniFlow [6], bringing together participants from MIT, MSU, and NCSU, to prototype a design flow demo of distributed tools residing on servers at MIT (CollabTop), MSU (JavaCADD), and NCSU (Xact) [8].

The current workflow technologies address the configuration problem mostly from the perspective of the workflow designer rather than the workflow user. Once accessed by the user, the domain-specific workflow supports executions of sequences of predefined tasks, e.g. [9, 10]. The underlying schemas of such workflows are too complex to support re-configurability by the average user [11, 12]. On the other hand, the *universal client (OmniFlow)* introduced in this paper, supports distributed networked design and computing *and* is also *readily reconfigurable by the user*. The ease of reconfigurability is achieved by the simplicity of the underlying *taskflow schema* based on *taskflow-oriented programming*: well-defined encapsulation and composition of only two types of software components, a blackbox and a whitebox.

The paper is organized into several sections as follows: (2) Background and Motivation; (3) Taskflow Architecture; (4) Taskflow Programming; (5) Conclusions.

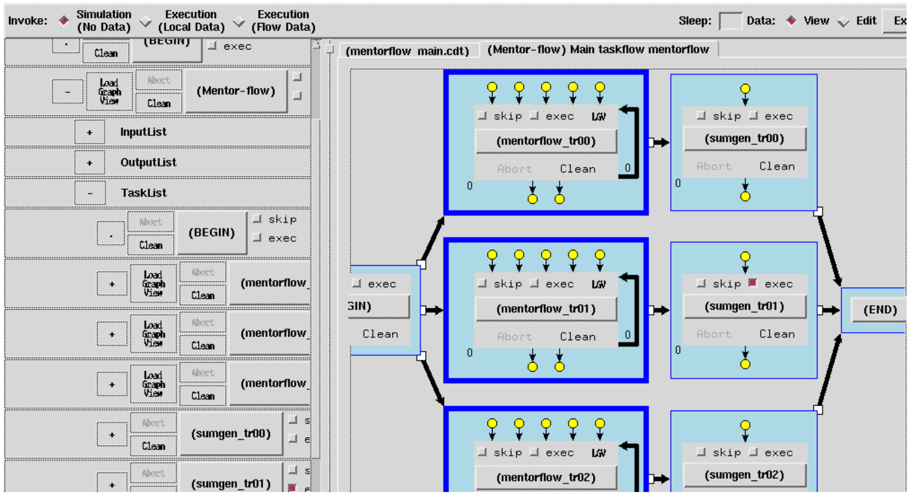
*This work has in part been supported by the contract from DARPA/ARO (DAAG55-97-1-0345).

†Hemang Lavana performed this work while affiliated with NC State University.

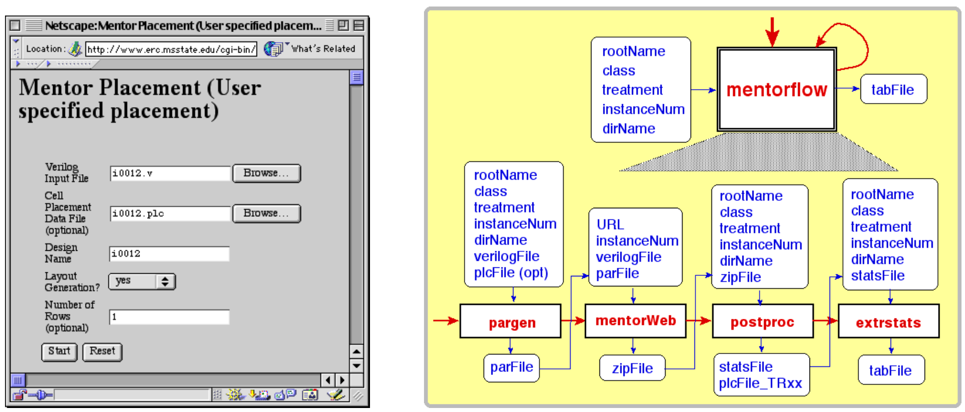
DAC 2001 June 18-22, 2001 Las Vegas, Nevada USA

2. BACKGROUND AND MOTIVATION

We use a simple example to illustrate a typical GUI created by the OmniFlow client. The example introduces a simplified *experimental design environment* whose purpose is to evaluate the performance of distributed algorithms with distributed participants. A comprehensive description of OmniFlow project drivers, including the one in Figure 1, is given in [7]; related experiments with an earlier OmniFlow version are also described in [6].



A *state* of a typical GUI created by the OmniFlow client. While features of the selector panel at the top do not change, the tree view panel on the left, and the graph view panel on the right are generated dynamically, in response to user interaction. As described in the paper, user interacts with objects in each panel, e.g. buttons to invoke/abort a task, circles to view/edit data associated with each task, and directed edges (that close/open) to control task execution sequence, including iteration, or to induce task aborts. Synchronization of tasks that execute concurrently is implicit in the XML-based task encapsulation.



The hierarchical taskflow in this example modifies the distributed MSU-NCSU experiment initiated in [6] and described in [7]. Rather than invoking the placement tool manually through a web-browser for each of the three placement algorithms and netlist instances, a *mentor flow description in OmniFlow* renders three instances of *mentor flow*, each scheduled to repeat concurrently N-times for different netlist instances from each of the M netlist equivalence classes.

Figure 1: An OmniFlow rendering of a hierarchical, distributed taskflow, executing concurrent tasks.

OmniFlow creates the GUI by rendering an XML description of the taskflow, with each task representing an instance of an encapsulated blackbox component or an encapsulated whitebox component. The blackbox component can be any (legacy) program on the Internet that is accessible via the TCP protocol using telnet-, ssh-, http-, or socket-based clients. The whitebox component is represented as a directed graph of blackbox and whitebox components. The OmniFlow client thus provides a *programmable* taskflow-oriented programming *and* computing environment that is highly interactive, as we will show shortly.

A single view of a taskflow with many components is difficult to represent clearly. The GUI in OmniFlow consists of three main panels, shown in Figure 1: *selectors* for program execution and data viewing/editing as the panel at the top; dynamically expandable *tree view* of the entire taskflow hierarchy as the panel on the left; and the dynamically generated *graph view* of the taskflow at any level of hierarchy on the right. A log message panel at the bottom is not shown.

Selector Panel. Using the controls in the selector panel, user can execute the taskflow in any of the three modes: *simulation, execution with local data, execution with flow data.* The simulation mode allows the user to execute the entire taskflow structure without specifying any data dependencies between tasks, with each task assigned a random variable to 'sleep' for a few seconds. Alternatively, user can enter fixed time to 'sleep' in the selector panel. This mode is useful to set-up and test the taskflow control structure as specified in user-defined *TaskGraph* description, including the verification for concurrent execution. The execution with local data is useful when verifying the performance of each task in the taskflow in a stand-alone context, with originally archived test data for each task. The execution with flow data implies that each task relies on data that may be generated dynamically by other tasks, as specified in user-defined *DataGraph* description. Clicking on the selector for data editing, user can view and edit a data file associated with a given task.

Tree View. Upon invocation, only the main task instance is displayed as the root of the tree view. The children of the main task instance can be opened or closed by clicking on a '+' or a '-' symbol located near the task instance node. On opening the main task instance, it displays the data I/O, if any, of the main task, such as *InputList*, *InOutList*, *OutInList* and *OutputList* and also its *TaskList*. Each TaskList can be expanded similarly until we reach the task represented by the blackbox component. 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, in the tree view, we do not see the *task-to-task*, *data-to-task* and *task-to-data* dependencies explicitly – hence the need for the *graph view*.

Graph View. As shown in Figure 1, each encapsulated task instance is represented as a button widget (a single click will invoke the instance), surrounded by (a) user-clickable control fields including an (optional) *repeatInvocation* edge, (b) circles and directed *data-edges*, connecting I/O files and variables with associated task I/O ports, (c) task boundary which is bold if the task encapsulates a whitebox, and thin if the task encapsulates a blackbox. The control fields include: 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 whitebox component; 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 `repeatInvocation` edge which is rendered in the 'connected' state if and only if user specified the repeat task condition in the taskflow description. However, user may subsequently decide to 'open' this edge by clicking on it, and close it again later with another click.

At any level of taskflow hierarchy, we capture the task instance dependencies in two directed graphs: *DataGraph* and *TaskGraph*. The *data-edges* in DataGraph connect I/O files and variables represented as circles with associated task I/O ports. By clicking on the circle, each file and variable can be accessed for viewing or editing. We can consider the view depicted in Figure 1 as an explicit representation of data used and generated in the mode labeled as 'execution with local data'. However, the actual description entered by the user as DataGraph consist of *data-to-task* and *task-to-data* edges *between* different tasks. To avoid clutter, these edges are not shown explicitly.

Dependencies that are shown explicitly in the graph view are *control edge* dependencies and they are of three types: *singleInvocation*, *repeatInvocation*, and *abortInvocation*. Only the first two types are shown in Figure 1. By supporting abortInvocation edges, taskflow synchronization can be rendered more efficient and versatile. Consider the situation where a task B is to proceed as soon as k-out-of-n tasks A_i have completed. By setting up the taskflow with n abort edges from B to each of A_i, we can abort the completion of the remaining $n - k$ tasks A_i that would otherwise still be executing and their results never used upon completion. By maintaining the distinct control edge types, we can maintain the TaskGraph as a directed acyclic graph that is also polar, with the 'begin' and 'end' task nodes at each level of hierarchy. As such, the TaskGraph can be readily scheduled for concurrent and sequential execution of all tasks. Once the edges are rendered in the taskflow graph view, user can interact with them by clicking on them to open/close any of them, thereby interacting with the taskflow scheduling engine as needed. A total of 8 states is associated with each task instance, and during execution, the 'white boxes' associated with the head of each singleInvocation edge changes color to indicate the current state of the task.

The selector panel, the tree view, and the graph view of the taskflow hierarchy provide an interactive and uniform environment not only for creating versatile taskflow structures but also for creating an environments that lend themselves to collaborative computing projects [7]. In the sections that follow, we formalize the taskflow architecture and highlight taskflow programming projects that test the scalability of this environment.

3. TASKFLOW ARCHITECTURE

In contrast to hardware components, the notion of a component in software technology is no simple matter. In [14], a chapter entitled *What a component is and is not*, provides an authorative analysis on the subject. Not surprisingly, notions of blackbox, whitebox, and encapsulation used in this paper have context that is specific to the proposed taskflow architecture [7].

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. We may deduce also the completion status by comparing time-stamps of input and output data sets.

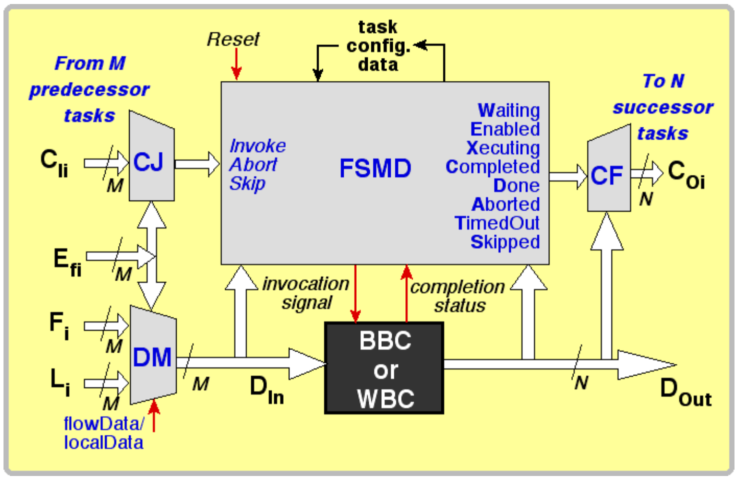
An *encapsulated blackbox component* is a finite-state- machine (FSM) arrangement with a blackbox component, where the blackbox component is an extension of the data path, communicating with the FSM by way of two *handshaking signals*. A finite-state-machine with a data path (FSMD) is common in high-level synthesis and hardware design. [13]. The blackbox is invoked by the companion FSMD, which in turn is invoked by the user or another program.

Whitebox component (Requirements). Informally, a whitebox is a composition of blackbox *and* whitebox component instances that support:

- creation of task sequences that execute sequentially as well as concurrently;
- data-dependent decisions for a block of task sequences;
- data-dependent iterations for a block of task sequences;
- component encapsulation for a block of task sequences;
- single entry and single exit point for each encapsulated component.

The proposed task instance architecture satisfies these tenets.

Task Instance Architecture. This architecture is based on an arrangement of five abstract task primitives: Finite-State-Machine with a Datapath (FSMD), ControlJoin (CJ),



Default ControlJoin conditions:

$$
\begin{aligned}
P_I &= \prod_{m=1}^{M} E_{f_m} \cdot Q_{V_m} \\
P_S &= \sum_{m=1}^{M} E_{f_m} \cdot (Q_{N_m} \\
&= \qquad + \; Q_{S_m} + Q_{T_m}) \\
P_A &= \sum_{m=1}^{P} E_{a_m} \cdot Q_{V_m}
\end{aligned}
$$

(For other join conditions, see [7].)

Data multiplexer:

E_g	E_{f_i}	D_{I_i}
0	x	D_{Il_i}
1	0	D_{Il_i}
1	1	D_{If_i}

The architecture of *each* task instance is based on the arrangement of *five* abstract task primitives: 8-state Finite-State-Machine with Datapath (FSMD), ControlJoin (CJ), DataMultiplexor (DM), ControlFork (CF) and a BlackBox or a WhiteBox component (BBC/WBC). The functional descriptions of the default CJ and the nominal DM are given below. The descriptions of other CJs, CF, and FSMD are given in [7].

The purpose of CJ is to synchronize the status of predecessor tasks before invoking the current task instance. The three possible signal pulses generated by CJ are: (1) an invocation pulse P_I, (2) a skip pulse P_S, or (3) an abort pulse P_A. The decision variables are: E_{f_m} (user-configurations of the singleInvocation control edge from task m represented by open/closed), E_{a_m} (user-configurations of the abortInvocation control edge) Q_{V_m} ('valid' task status), Q_{N_m} ('invalid' task status), Q_{S_m} ('skip' task status), and Q_{T_m} ('time-out' task status). For example, pulse P_I is generated if all singleInvocation control edges are closed and the task status associated with each edge is 'valid'.

The purpose of DM is 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_{Il_i}. Note that DM selects the flow data only when E_g and E_{f_i} are *both* enabled!

Figure 2: The architecture of the task instance.

DataMultiplexor (DM), ControlFork (CF) and a blackbox/ whitebox component, as shown in Figure 2. While the arrangement of FSMD and blackbox/whitebox component alone represents the component encapsulation, each task instance represents an encapsulated component that can be accessed by other components and data only via ControlJoin, DataMultiplexor, and ControlFork. These primitives represent combinational logic and can be described in terms of Boolean equations or tables; examples are shown in Figure 2.

The purpose of the ControlJoin is to synchronize the status of predecessor tasks before invoking the current task instance. A number of such conditions may exist, depending on the purpose of the current task; a representative set of alternative ControlJoin conditions is listed in [7].

The purpose of ControlFork primitive is not only to output the state of the FSMD when the task completes but also to validate it against the user-specified condition, if any. For example, if the user-specified condition for task k is '$size(D_{O_k}) > 128$', a 'valid' state Q_{V_k} is generated when the condition evaluates to true.

The FSMD primitive is at the very core of the proposed task instance architecture and is described in terms of a state-transition table and a datapath table [7]. Scheduling the invocation of a task instance T_k is subject to evaluation of a number of control signals as well as data values. All evaluations take place within the ControlJoin, FSMD and ControlFork associated with the task instance T_k [7].

In choosing the FSM model to encapsulate each component, we show preference for the traditional (and relatively simpler) hardware-based solutions over alternative approaches that may rely on the formalisms of Petri Nets, Actor Computations, Action Systems, etc. (see [7] for citations). 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.

TaskFlow Schema. The interconnection of the task primitives such as defined earlier is subject to few simple and well-defined rules. Each of the interconnection rules defines a taskflow layer. In effect, the task instance architecture in Figure 2 is the basis for the *task instance layer* which encapsulates a task, which in turn may be composed of other task instances, each of which encapsulates a task, etc. Subsequently, a schema to construct a taskflow consists of two principal layers, a single/multi-task encapsulation layer and a task instance layer. The multi-task encapsulation layer represents an encapsulated whitebox component in which at least one task instance is invoked by a ControlFork primitive and at least one task instance invokes a ControlJoin primitive. For clarity, we rename these primitives as *Begin-Fork* and *EndJoin* whenever we discuss the encapsulation layers.

The structure of the proposed taskflow schema is shown in Figure 3, along with illustrative examples of TaskGraph and DataGraph descriptions in XML. The definition layer can be considered as an API for the task and should be readily accessible. The task body layer, as the name suggests, contains more detailed information about the task specifics. Details, including the complete taskflow schema in XML, are available in [7].

TaskFlow Scheduling Engine. See Figure 4 for a brief introduction and overview of the taskflow scheduling engine.

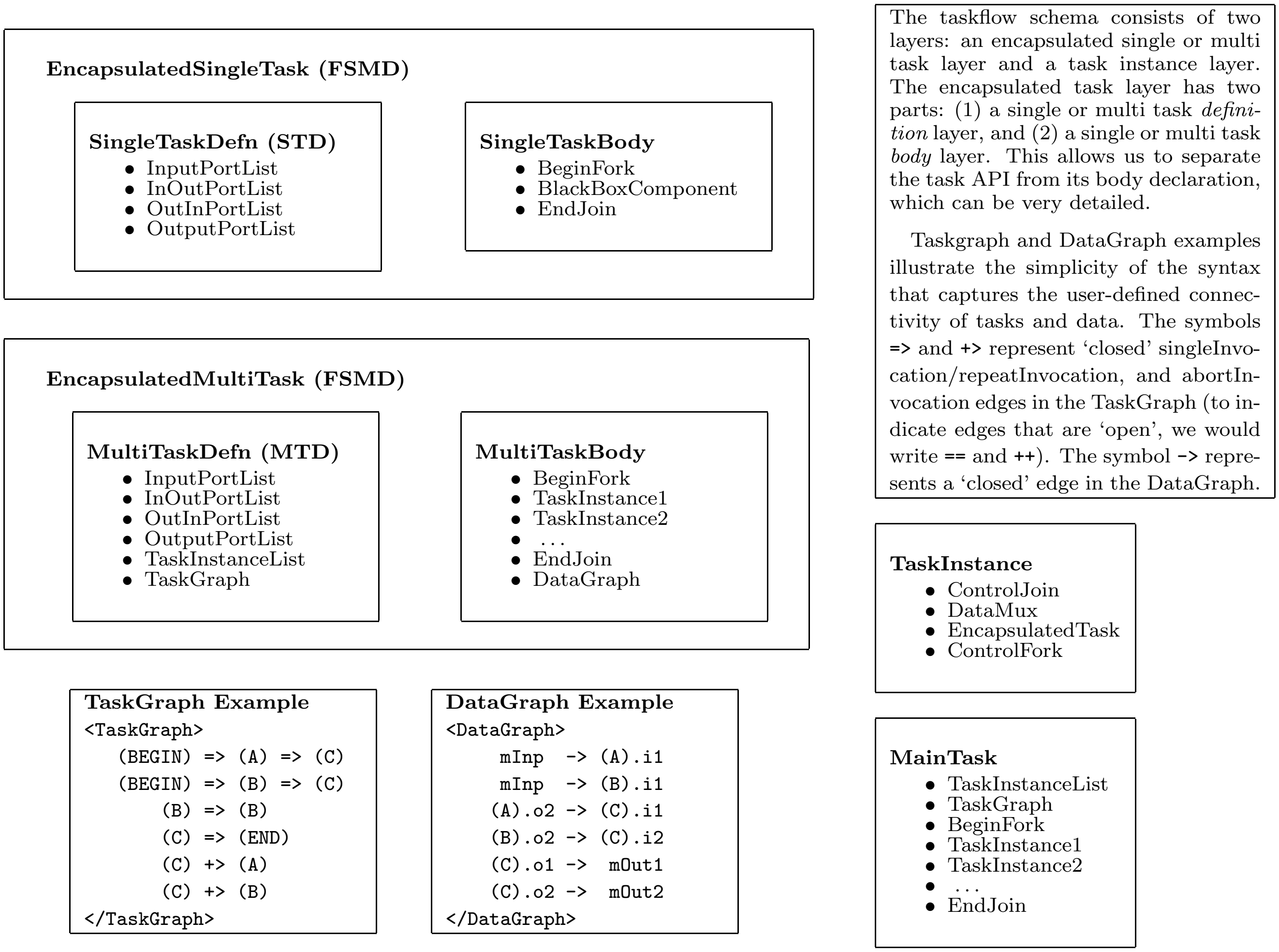


Figure 3: A schema for XML representation of taskflow layers.

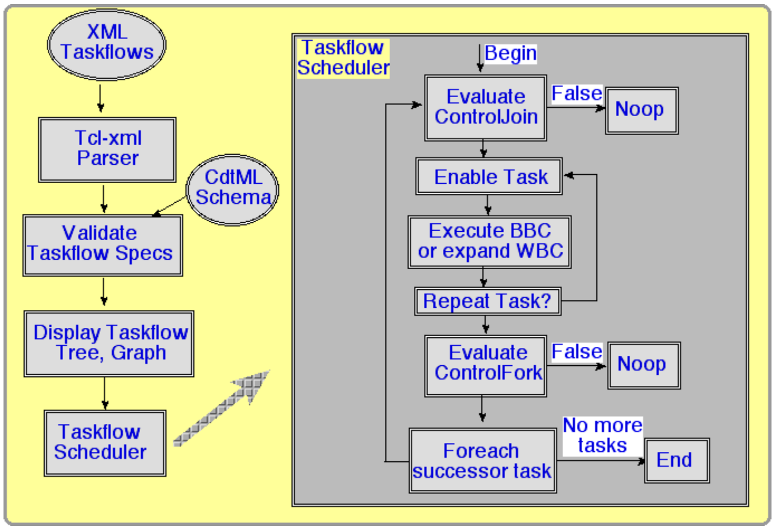
4. TASKFLOW PROGRAMMING

Taskflow programming enables the user to create a highly interactive and executable 'program-of-programs' that invokes component programs accessible on the Internet via telnet-, ssh-, http-, or socket-based clients. The program represents a project-specific configuration of the OmniFlow client, and is rendered as a hierarchy of executable taskflows. To create a taskflow such as described in Figure 1, the user may proceed as follows:

- create a file, containing a short 'main' program, that invokes the taskflow.
- create a file, about 8 definition blocks for each task instance, as per Figure 1. At this point, the taskflow can be invoked in simulation mode to test its control structure at all levels of hierarchy.
- create a file containing the 'body' corresponding to each of definitions, including all pointers to hosts and data directories. Any input synchronization, output validation, and task iteration conditions must be stated – unless user relies on built-in defaults. Once the program is tested on smaller test examples, data may be prepared for a major batch of executions, and the program re-invoked.

Writing such descriptions is not unlike writing a description in a hardware description language at the structural and the behavioral level. The advantages of writing the description as an OmniFlow configuration are: (1) the near-instant rendering of the taskflow hierarchical structure (not always obvious from the textual description) – before capturing any of the taskflow body description and data; (2) dynamic testing of the control structure of the taskflow description – before capturing any of the taskflow body description and data; (3) practically unlimited freedom to reconfigure, via a standardized GUI, the flow of execution dynamically, once the taskflow body description and data are rendered.

Comprehensive experiments with a variety of taskflow demonstrate the efficiency of the GUI implementation and the scheduler [7]. Rendering a taskflow such as shown in Figure 1 is on the order of 2 seconds (under Solaris/Linux/WindowsNT/MacOS). Large-scale experiments with taskflow configurations ranging from 15 to 9150 task instances, with longest path delay of 1600 tasks, reveal a near constant overhead of processing each task, independent of time to execute the task and also independent of the structure of the taskflow. For example, the overhead per task in a taskflow of 2400 instances where most task are executed sequentially (longest path delay is 1600 tasks) and a taskflow of 2400 instances where most task are executed concurrently (longest



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.

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 prompted only by (1) user request to load/view the task instance in the GUI, or (2) invocation of a task instance scheduled for execution. The taskflow scheduler consists of a recursive algorithm which calls itself for every occurrence of the whitebox instance in the taskflow.

The dynamic loading of task instances and runtime scheduling of task execution results in an efficient and scalable implementation, demonstrated on a progression of taskflows of increasing size [7].

Figure 4: Implementation architecture of taskflow scheduling engine.

path delay is 100 tasks) is as follows:

taskflow(2400/1600): 922.8/2400 = 0.384 seconds/task
taskflow(2400/100): 979.5/2400 = 0.408 seconds/task

For taskflows with 300 instances, the overhead amounts to:

taskflow(300/200): 123.6/200 = 0.412 seconds/task
taskflow(300/100): 118.8/300 = 0.396 seconds/task

Given that most tasks may require a number of seconds to complete, the taskflow overhead is negligible.

5. CONCLUSIONS

Hardware description languages and structured methodologies play a major role in supporting the productivity and advances in the design of complex systems that package millions of transistors onto a single IC – not individually, but as large components. In this paper, we introduce the concept of taskflow-oriented programming with distributed components and a highly interactive universal client GUI as a new paradigm to create configurable computing environments for distributed and networked design projects.

A user guide and a cross-platform software prototype of the client described in this paper will be posted by mid-June 2001 under

`http://www.cbl.ncsu.edu/OpenProjects/OmniFlow/`.

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

6. REFERENCES

[1] Toolwire, 2000. See `http://www.toolwire.com`.

[2] H. Lavana, A. Khetawat, F. Brglez, and K. Kozminski. Executable Workflows: A Paradigm for Collaborative Design on the Internet. In *Proc. of 34th DAC*, pages 553–558, June 1997. See `http://www.cbl.ncsu.edu/-publications/#1997-DAC-Lavana`.

[3] D. Linder, R. Reese, J. Robinson, and S. Russ. JavaCADD: A Java-based Server and GUI for Providing Distributed ECAD Services, April 1998. Technical Report MSSU-COE-ERC-98-07. See `http://WWW.ERC.MsState.Edu/mpl/publications/-papers/javacadd/JCaddTR.pdf`.

[4] F. Chan, M. Spiller, and R. Newton. WELD - An Environment for Web-Based Electronic Design. In *Proc. of 35th DAC*, pages 146–152, June 1998.

[5] G. Konduri and A. Chandrakasan. A Framework for Collaborative and Distributed Web-Based Design. In *Proc. of 36th DAC*, June 1999.

[6] H. Lavana, F. Brglez, R. Reese, G. Konduri, and A. Chandrakasan. OpenDesign: An Open User-Configurable Project Environment for Collaborative Design and Execution on the Internet. In *Proc. of Intl. Conference on Computer Design*, September 2000. See `http://www.cbl.ncsu.edu/-publications/#2000-ICCD-Lavana`.

[7] H. Lavana. *A Universally Configurable Architecture for Taskflow-Oriented Design of a Distributed Collaborative Computing Environment*. PhD thesis, Elec. & Comp. Eng., NCSU, Raleigh, N.C., December 2000. See `http://www.cbl.ncsu.edu/publications/-#2000-Thesis-PhD-Lavana`.

[8] F. Brglez. OpenProjects Home Page, with links to DAC'2000 Vela Project Demos and Software., June 2000. See `http://www.cbl.ncsu.edu/OpenProjects/`.

[9] Iflow: Internet Enabled Workflow, 2000. See `http://www.justdotit.com.au/html/-iflow_frame.htm`.

[10] ActionWorks Metro: Web-based workflow software, 2000. See `http://www.actiontech.com/Action/`.

[11] XML Schema Home Page, September 2000. See `http://www.w3.org/XML/Schema.html`.

[12] S. Jablonski and C. Bussler. *Workflow Management Modeling Concepts, Architecture and Implementation*. Thomson Computer Press, 1996.

[13] D. Gajski and A. Wu and N. Dutt and S. Lin. *High-Level Synthesis Introduction to Chip and System Design*. Kluwer, 1992.

[14] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison Wesley, 1998.