A Universal Client for Taskflow-Oriented Programming with Distributed Components: Concepts [*]

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

This paper introduces the concept of taskflow-oriented programming by way of a universal, configurable client that (1) reads *user-programmed interconnections* of distributed component programs as task instances, (2) dynamically renders an *interactive GUI* of all interconnected components as a hierarchical taskflow, and (3) dynamically creates a schedule to execute component programs concurrently, serially, or not at all, depending on the user-defined runtime configuration of the taskflow topology. The implementation of the client, including the transparent access to components via a TCP protocol using telnet-, ssh-, http-, or socket-based clients, is presented in the companion paper.

Conceptually, taskflow-oriented programming relies on a recursive schema of encapsulated blackbox (whitebox) component instances. Each encapsulated component instance contains five primitive tasks: a blackbox (whitebox) component, an eight-state finite-state-machine with a datapath (FSMD), a ControlJoin, a ControlFork, and a DataMux. User-programmed interconnections of distributed component programs are captured in the ControlJoin and ControlFork of each component instance. The taskflow schedule is derived from the underlying TaskGraph of asynchronously interacting FSMDs, each supporting a simple hand-shaking protocol with the attached blackbox (whitebox) component.

1. INTRODUCTION

The re-use of software components can lead to rapid development of new software applications. In contrast to hardware components, the notion of a component in software technology is no simple matter. In [1], a chapter entitled *What a component is and is not*, analyzes a total of 10 definitions. A definition formulated as one outcome of the Workshop on Component-Oriented Programming is the following [2]:

> *'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.'*

In reality, the components emerging today are based on standards that may compete and conflict with each other: OMG's CORBA [3], Sun's Java [4], Microsoft's COM [5]. A software component considered in this paper has the granularity of a *megamodule* [6]:

> *'Megamodules are internally homogeneous, independently maintained software systems [....] Each megamodule describes its externally accessible data structures and operations and has an internally consistent behavior.'*

i.e. such components are stand-alone programs, installed and maintained on a specific host on the network.

In contrast to developing a megamodule composition, programming architecture, and language/compilation environment as envisioned in [6], the widely practiced approaches to composition of stand-alone modules into a single program rely on scripting languages [7]. Scripting is widely used by programmers to deliver packaged multi-component applications whose interface may range from a simple command-line to complex GUI in workflows, e.g. [8, 9] – and is typically *not* user-configurable. Such programs require expert programming effort and expert software maintenance, especially if components are distributed across heterogeneous hosts and file systems, support not only serial but also concurrent execution, and are expected to facilitate collaboration between distributed teams to access shared data and to control invocation of specific components.

When digital system designers create component-based systems, they routinely use a digital *simulator client* to analyze the behavior of a specific configuration of interconnected components, and if desired, invoke another client to map the configuration onto a hardware module. Such clients are written by a relatively small programming teams – but for a large number of client users. In contrast, when it comes to creating a new program from existing software components, the prevalent approach today is to engage an expert programmer to write a customized script each and every time. In this and the companion paper [10], we propose an alternative: *a taskflow-oriented programming paradigm*, with task instances representing distributed stand-alone component programs, such that *users*, without assistance from expert programmers, can compose interactive, executable programs using these components. Such compositions can be also rendered collaborative if and when needed. This

[*]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.

<u>*Task “Place & Route & Simulate” ....*</u>

Task A1: place standard cells from netlist $N(a)$ @host a1
Task A2: place standard cells from netlist $N(a)$ @host a2
Task B1: place blocks from netlist $N(b)$ @host b1
<u>*Task C1:*</u> route all placed netlists ...
  Task C1a: abort 3rd placement, download 2-of-3 placements @host c1
  Task C1b: route both placements @host c1
<u>*Task D1:*</u> annotate/simulate p&r netlists ...
  Task D1a: annotate netlist $N(a)$ with extracted layout @host c1
  Task D1b: annotate netlist $N(b)$ with extracted layout @host c1
  <u>*Task D1c:*</u> simulate annotated netlists ...
    Task D1c1: pre-process input data @host c1
    Task D1c2: simulate @host c1
    Task D1c2: post-process output data @host c1

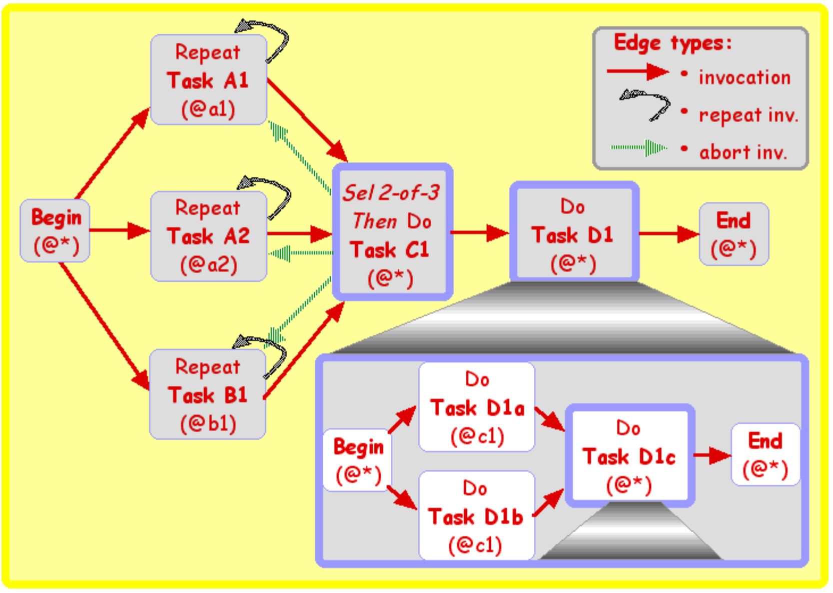


Figure 1: A tree and a graph view of taskflow hierarchy, represented as blackboxes and whiteboxes.

approach has a direct analogy with the ‘digital simulator’ client, which is created once for a large class of applications and users. Rather than engaging a programmer to write a custom script each time a composition of a new software system is needed, *user alone* can now (1) write a taskflow configuration, (2) invoke the *universal client (OmniFlow)* that reads the configuration and renders it as a highly interactive GUI, and (3) interact with the taskflow in any of the following ways: reconfigure the taskflow interconnections, view or edit input/output data, descend/ascend the taskflow hierarchy, select the mode of execution, invoke the taskflow, abort the taskflow (if already executing), reset the state of the taskflow, etc.

Conceptually, taskflow-oriented programming relies on a recursive schema of encapsulated blackbox (whitebox) component instances. Each stand-alone component is represented as a blackbox component; a whitebox is simply a composition of blackbox and whitebox components. Each encapsulated component instance contains five primitive tasks: a blackbox (whitebox) component, an eight-state finite-state-machine with a datapath (FSMD), a ControlJoin, a ControlFork, and a DataMux. User-programmed interconnections of distributed component programs are captured in the ControlJoin and ControlFork of each component instance. The taskflow schedule is derived from the underlying TaskGraph, a directed polar graph of asynchronously interacting FSMDs, each supporting a simple hand-shaking protocol with the attached blackbox (whitebox) component. Not surprisingly, notions of a blackbox, a whitebox, and an encapsulation used in this paper have a context that is specific to the proposed task instance architecture, first introduced in [11].

The paper is organized into several sections: (2) Background and Motivation, (3) Taskflow Architecture that introduces TaskGraph, DataGraph, FSMD, ControlJoin, ControlFork, and DataMux, (4) Taskflow Schema and Scheduling, (4) Representative Taskflow Patterns, and (5) Summary and Conclusions.

2. BACKGROUND AND MOTIVATION

To a large extent, two project drivers motivated the development of the universal user-configurable client: one involved a prototype testbed for distributed experimental design and performance evaluation of graph-based algorithms [11, 12], the other a distributed VLSI design flow environment consisting of commercial and university-based tools, some residing at MSU, some at MIT, and some at NCSU [11, 13].

At a first glance, the two projects would not appear to have much in common. However, when we organize the computational tasks involved in each of the projects, two very generic views about such organizations emerge. One is a *tree view*, where the computational project is organized as a hierarchy of tasks in a rooted tree. The second one is a *graph view* that intuitively depicts the choices of sequences in which tasks may be invoked and executed. An illustrative example of such representations, based on a realistic segment of VLSI design tasks, is shown in Figure 1 and can be described in few words as follows:

- invoke, on a local host (@*), the task node named *Begin* to *concurrently* invoke three tasks $\{A1, A2, B1\}$, each representing a placement algorithm executing on hosts $\{a1, a2, b1\}$, given a netlist $N(a)$ as data input under host $\{a1, a2\}$ and a netlist $N(b)$ as data input under host $\{b1\}$. Tasks *may* be invoked concurrently since all have been assigned to unique hosts. Each of these three tasks may be repeated a number of times, subject to specific terminating conditions. Here, each task represents an instance of an encapsulated blackbox, i.e. a stand-alone program with a standardized interface.
- invoke task $C1$ as soon as either of task pairs $\{A1, B1\}$ or $\{A2, B1\}$ completes last iteration. This task represents a whitebox instance, composed of two blackbox instances, invoked on host $c1$ serially. First, one of the still executing tasks, $A1$ or $A2$, is aborted, and two placement files along with netlists $N(a)$ and $N(b)$ are

downloaded to host $c1$. Next, given the (x, y) coordinates of all placed objects and the respective netlists, all pin-to-pin connections are routed to complete the task $C1$.

- invoke task $D1$ as soon as task $C1$ completes. This tasks represents a whitebox instance, composed of two blackboxes, $D1a$ and $D1b$, and a whitebox $D1c$. Since $D1a$ and $D1b$ are assigned to the same host, they are executed serially in random order. Otherwise, they would have been executed concurrently. As soon as *both* of these tasks complete, task $D1c$ is invoked, expanding into tasks $\{D1c1, D1c2, D1c3\}$ for serial execution, as determined by the task dependencies.

The graph view in Figure 1 is in fact a representation of a *TaskGraph*, a hierarchical directed polar graph with three edge types (invocation, repeat invocation, and abort edge). The nodes designated as *Begin* and *End* are the source and the sink task nodes that make the graph a polar graph. The taskflow itself is an intersection of a *TaskGraph* and a *DataGraph*, to be discussed in greater detail in the next section.

The distinct representation of the taskflow edges is useful for both the visual interface as well as the internal representation. Whenever a task is to be repeated, a repeat invocation edge is present to indicate the intention. Whenever a task is to be *potentially aborted*, an abort invocation edge is present. Given the task structure and the task synchronization as described for Figure 1, the omission of abort edges would result in a less efficient implementation of this taskflow, since one of the tasks, $A1$ or $A2$, would have been executing to its scheduled completion with no chance to impact on the overall taskflow objectives.

2.1 Taskflow GUI Features

Both the tree and the graph view in Figure 1 serve as the basis for the graphical-user interface (GUI) described in the companion paper [10]. There is a wide range of possible user interactions with this interface, supported by the taskflow architecture as described in the section that follows. To introduce representative elements of the proposed taskflow-oriented programming paradigm, we list and elaborate on a number of features of the taskflow GUI:

(1) structured text-based entry of the taskflow hierarchy, with each taskflow represented as an intersection of a TaskGraph and DataGraph. This includes user-entry of location, access protocol, and invocation privileges for each blackbox component being encapsulated.

(2) automatic generation of the GUI from user-entered textual description of encapsulated task instances.

(3) point-and-click open, close, ascent, and descent of the tree and the graph hierarchy.

(4) point-and-click reconfiguration of any invocation, repeat invocation, and abort invocation edge into a 'closed' and 'open' state.

(5) point-and-click invocation of the taskflow and its schedule by clicking on any of its task nodes, including the *Begin* task node. The schedule is generated dynamically, relative to the invoked task node, and subject to the user-defined runtime configuration of all edges. Tasks *may* be invoked only if driven by invocation edges that are in 'closed' state.

(6) point-and-click abort of the taskflow by clicking on one or more of the executing task nodes, propagated in a descending order of taskflow hierarchy.

(7) point-and-click reset of the taskflow state, propagated in a descending order of taskflow hierarchy.

(8) point-and-click access to view and edit data associated with each task, represented as input (output) data nodes associated with each task (such nodes are shown in Figure 2 but not in Figure 1).

(9) point-and-click execution of the taskflow using *local data* for each task rather than *flow data* generated dynamically by other tasks in the flow.

(10) point-and-click execution (and testing) just the control structure of the taskflow, using no data.

To support the features of the proposed taskflow GUI, a taskflow architecture must be formalized. We outline requirements for such an architecture next.

2.2 Taskflow Architecture Requirements

The *interfaces* between the task instances as depicted in Figure 1 are dominated by three types of directed task-to-task control edges: *InvocationEdge, RepeatInvocationEdge,* and *AbortInvocationEdge*, whose enabled/disabled (closed/open) state is also under user-control. While data-to-task and task-to-data edges are not shown explicitly, it is understood that the implicit placement of these edges induces task-data and data-task dependencies that are compatible with the explicit placement of the control edges as shown. For example, task instances $\{A1, A2, B1\}$ in Figure 1 are independent of each other; all can be invoked and executed concurrently since none depend on output data of the other.

The interfaces, not seen in the GUI, between the task instance layer and the layer of each (unencapsulated) blackbox itself are central to the taskflow architecture definition as described in the next section. The taskflow architecture requirements are rooted in the principles, the discipline, and the simplicity of structured programming composition schemes. The truly fundamental of these schemes are, with respect to each task:

sequencing, conditioning, and replication[1].

All of these composition schemes are illustrated in the graph view in Figure 1. Moreover, another essential composition scheme, also embodied in this graph view, relates to 'manageability of programs' as eloquently articulated in [14]:

> *Our most important mental tool for coping with complexity is* abstraction*. . . . For the intellectual manageability, it is crucial that the constituent operations at each level of abstraction are connected to sufficiently simple program schemas, and that each operation is described as a piece of program with* one starting point *and* a single termination point. *This allows defining states of the computation* (P, Q)*, i.e. relations among the involved variables, and attaching them to the starting and terminating points of each operation* (S)*. It is immaterial, at this point, whether these states are defined by rigorous mathematical formulas (i.e. by predicates of logical calculus) or*

[1]Here, we paraphrase [14], where such composition schemes are described with respect to program statements.

by sufficiently clear and informative sentences, or by combination of both. The important point is that the programmer has the means to gain clarity about the interface conditions between the building blocks out of which he composes his program [15].

An example of operation (S) is any whitebox component in Figure 1, with embedded starting and terminating points represented as *Begin* and *End* task primitives, redefined formally as *BeginFork* and *EndJoin* in the next section. Major requirements that are to be supported by the taskflow architecture as proposed in this paper are thus as follows:

(1) each task instance encodes the state of outgoing InvocationEdges, RepeatInvocationEdges, and AbortInvocationEdges whether to invoke the incident task once, repeatedly, or whether to abort the incident task being executed.

(2) the state of each edge is read by the incident task if and only if the user has enabled the edge to be 'closed' rather than 'open'.

(3) each task instance decodes only the enabled states of incoming InvocationEdges, RepeatInvocationEdges, and AbortInvocationEdges. By default, a task is invoked if and only if the state of *all enabled* InvocationEdges is *valid*. A number of other special task synchronization conditions can be considered, such as the one shown for task $C1$ in Figure 1, and others discussed in Section 3.2 of the paper. A task is repeated if and only if the state of its RepeatInvocationEdge is *enabled* and *valid*. A task is aborted if it is in an executing state and the state of at least one of its AbortEdges is *enabled* and *valid*.

(4) each task instance may synchronize two or more predecessor tasks and invoke concurrently one or more successor tasks.

(5) the primitive task *Begin* or *BeginFork* has no explicit incoming edges, only one or more outgoing InvocationEdges, and zero or more AbortInvocationEdges. The outgoing edges are enabled either by the task parent or via the point-and-click GUI. In either case, such enabling may be subject to data-specific conditions.

(6) the primitive task *End* or *EndJoin* has no explicit outgoing edges, only one or more incoming InvocationEdges, and zero or more AbortInvocationEdges. By default, the task is completed if and only if the state of *each enabled* InvocationEdge is *valid*.

We formalize the taskflow architecture in the next section. In subsequent sections, we introduce the taskflow schema, taskflow scheduling algorithm, and a number of taskflow synchronization patterns that are readily supported by the proposed taskflow-oriented programming paradigm.

3. TASKFLOW ARCHITECTURE

From a user perspective, a universal client that supports taskflow-oriented programming is expected to have a number of features, such as listed in the previous section. The taskflow architecture that supports an implementation of such a client is based on layering a few simple formal models. For clarity, we introduce these models in two stages of successive refinements:

- taskflow primitives, graphs and layers;
- task instance architecture.

3.1 Taskflow Primitives, Graphs, and Layers

Each taskflow rendered by the universal client can be decomposed into a total of five abstract *primitive task types*:

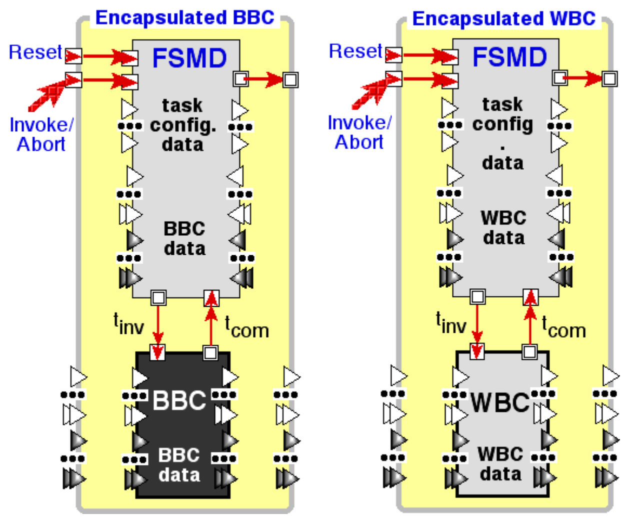
- *BlackBoxComponent (BBC)*;
- *FiniteStateMachine and Datapath (FSMD)*;
- *BeginFork/ControlFork (BF/CF)*;
- *EndJoin/ControlJoin (EJ/CJ)*, and
- *DataMux (DM, a data multiplexor)*.

BeginFork and ControlFork are functionally equivalent, so are EndJoin and ControlJoin. The naming convention will be clear once we describe the context in which each primitive task is used. Each task communicates with its environment by way of *directed edges*, attached to ports on the task layer. Ports hold, and can be probed, for two classes of variables: *control variables* or *data variables*. 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; data may be reused by the task on subsequent repeated invocations.

We first describe the role of each primitive in the context of its arrangement with other primitives and layers of hierarchy. Starting at the top-most level, we have *a taskflow layer* or *a whitebox component layer*, connecting a single instance of a BeginFork task primitive, any number of task instances, and a single instance of an EndJoin task primitive. Expanding any *task instance layer*, we always find an instance of a ControlJoin, a ControlFork, and a DataMux, connected to an encapsulated blackbox (whitebox) component. Expanding the *encapsulated blackbox (whitebox) component layer* further, we find a FiniteStateMachine and Datapath (FSMD) primitive connected to a blackbox (whitebox) component layer. The process of expansion repeats if the encapsulated component represents an encapsulated whitebox. The process terminates when all encapsulated components are encapsulated blackboxes. Examples of these relationships are illustrated in Figure 2. We now describe these layers in more detail.

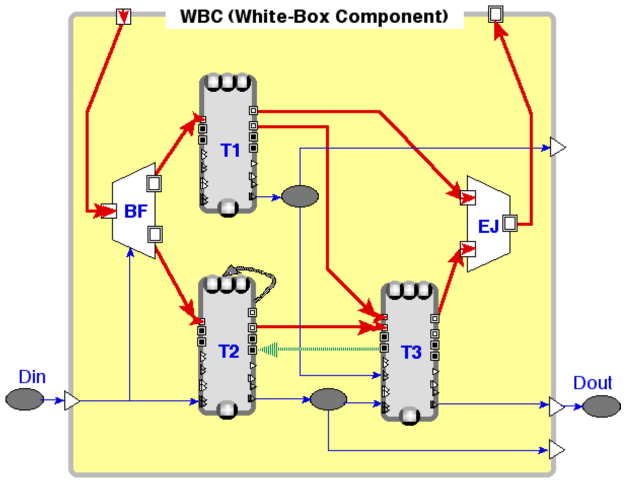
BlackBoxComponent (BBC) Layer. *A blackbox component (BBC)* is a stand-alone component program executable on a specific host. Its layer (that cannot be expanded any further) has an *invocation/abort control port*, a *status control port*, any number of *input data ports*, and any number of *output data ports*. A program may be invoked, and when executing, aborted via the same control port. When invoked and executing, the program may read input data, it may write output data, terminate and signify completion. By comparing time-stamps of input and output data sets, we may also deduce the completion status.

Encapsulated BlackBoxComponent Layer. *An encapsulated blackbox component layer* represents an arrangement of a BBC with a FSMD (finite-state-machine with datapath). Here, the blackbox component is simply an extension of the data path, communicating with the FSMD by way of a two *handshaking signals*: invocation (t_{inv}) and completion

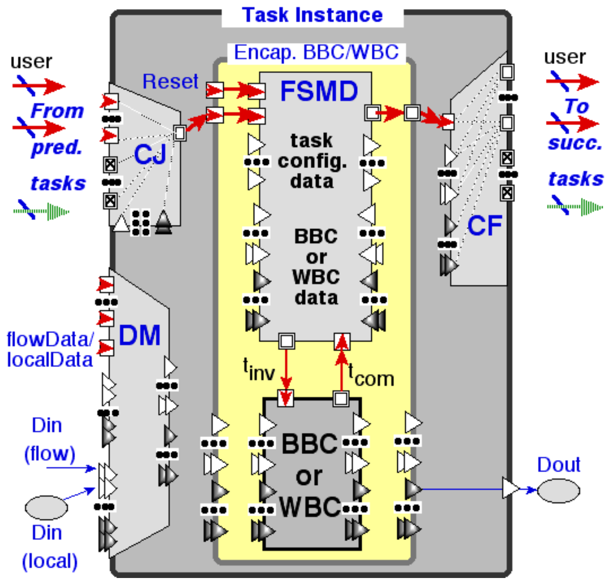


An encapsulated blackbox (whitebox) component layer represents an arrangement with a FSMD (finite-state-machine with datapath [16]). Control and data ports at the encapsulated layer are connected to control and data ports at the blackbox (whitebox) component layer. Connections by directed *internal control* edges are shown for two types of control ports at respective layers, the *invocation port* with an arrow and the *status port* with a square. Significantly, signals t_{inv} and t_{com} that *internally* connect FSMD to the respective blackbox (whitebox) component, illustrate that the blackbox (whitebox) component is simply an extension of the data path, communicating with FSM by way of the two handshaking signals.

There are two types of data ports, shown as triangles: ones to connect persistent data (white) and ones that connect temporary data (shaded). In addition, ports can be represented as port stacks if data is to be read and written in a specific sequence. For clarity, connections between data ports are not shown.



A whitebox component represents a hierarchical taskflow of directed acyclic graphs (DAGs) with nodes as (hierarchical) task instances and data instances. Each DAG is a polar graph, with *BeginFork (BF)* and *EndJoin (EJ)* primitive tasks representing the source node and the sink node respectively. Furthermore, each DAG can be represented as an intersection of *a TaskGraph* and *a DataGraph*. The nodes in the TaskGraph are task instances of blackbox (whitebox) components, including the BeginFork and EndJoin node, and there are three types of GUI-controlled directed edges: *InvocationEdge*, *RepeatInvocationEdge*, and *AbortInvocationEdge*. Each RepeatInvocationEdge can only create a self-loop with the task that is driving it. The DataGraph is a directed bipartite graph with two types of nodes: task instances as described in the TaskGraph and data instances, all connected with directed data-to-task and task-to-data *DataEdges*.



A task (a taskflow) instance layer is an arrangement of an encapsulated blackbox (whitebox) component and three task primitives: *ControlJoin (CJ)*, *ControlFork (CF)*, and *DataMux (DM,* data multiplexor), with data attached to its ports. For clarity, only connections between control ports of the encapsulation layer and the task instance layer are shown. Most significantly, the task instance layer is part of the GUI, along with the GUI-controlled InvocationEdges, RepeatInvocationEdges, and AbortInvocationEdges, and also with DataEdges to connecting tasks to data nodes. The purpose of the DataMux task is to select *local* or *flow* data before actually invoking the encapsulated component. Functionally, ControlJoin is equivalent to EndJoin task; its purpose here is to decode and synchronize the enabled states of incoming InvocationEdges before deciding whether to invoke the encapsulated component. Similarly, ControlFork is equivalent to BeginFork task; its purpose here is to validate the states of all outgoing InvocationEdges that may or may not invoke all successor task instances concurrently.

Figure 2: Layered primitives of the task model.

(t_{com}). Such arrangements are common in synthesis and design of hardware systems [16, 17]. The blackbox itself is invoked by the companion FSMD, which in turn is invoked by the user or another program via its own FSMD.

Whitebox Component (WBC) Layer. *A whitebox component (WBC)* contains two or more task instances. In general, a whitebox component represents a hierarchical taskflow of directed acyclic graphs (DAGs) with nodes as (hierarchical) task instances and data instances. Each DAG is a polar graph, with *BeginFork (BF)* and *EndJoin (EJ)* primitive tasks representing the source node and the sink node. Furthermore, each DAG can be represented as an intersection of *a TaskGraph* and *a DataGraph*. The nodes in the TaskGraph are task instances of blackbox (whitebox) components, including the BeginFork and EndJoin node, and there are three types of GUI-controlled directed edges: *InvocationEdge*, *RepeatInvocationEdge*, and *AbortInvocationEdge*. Each RepeatInvocationEdge can only create a self-loop with the task that is driving it. The DataGraph is a directed bipartite graph with two types of nodes: task instances as described in the TaskGraph and data instances, all connected with directed data-to-task and task-to-data *DataEdges*.

EncapsulatedWhitebox Component Layer. *An encapsulated whitebox component layer* represents an arrangement of a WBC with a FSMD (finite-state-machine with datapath). Just as in the case of blackbox encapsulation, the whitebox component is simply an extension of the data path, communicating with the FSMD by way of a two *handshaking signals* t_{inv} and t_{com}.

Task Instance Layer. *A task instance layer* contains a connection of three task primitives to the ports of the encapsulated blackbox or whitebox component: *ControlJoin* drives the invocation port of the FSMD and *ControlFork* is driven by the status port of the FSMD. A *RepeatCondition*, whether to repeat the task or not, is evaluated within FSMD. In addition, a *DataMux* selects between the two input data sets: a data set that typically is used to test the task instance in a local context and a data set generated by other tasks in the context of the taskflow execution. Only the selected data is accessible to the encapsulated task component.

3.2 Task Instance Architecture

A unique attribute of the taskflow architecture is its support for a recursive schema of encapsulated blackbox (whitebox) task instances. The encapsulation defines a layer that represents an arrangement of *an eight-state* asynchronous finite-state-machine and datapath (FSMD) with either a blackbox or a whitebox component. A whitebox component is a taskflow of two or more task instances. A task instance represents a layer that contains an encapsulated task (a blackbox or a whitebox component) connected to three primitive task components: *ControlJoin*, *ControlFork*, and *DataMux*. We now expand from the introductory description in Figure 2 to the architectural description in Figure 3.

The task primitives ControlJoin, DataMultiplexor, and ControlFork represent combinational logic and can be described in terms of Boolean equations. In Figure 3, we present the equations in a tabular form and 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 conditions may exist, depending on the purpose of the current task; a representative set of alternative ControlJoin conditions is listed in [10], more detailed in [11].

Asynchronous FSM with Datapath (FSMD). The abstract task primitive FSMD is at the very core of the proposed task instance architecture. In the current FSMD implementation, there are a total of eight states:

1. <u>W</u>aiting	5. <u>D</u>one
2. <u>E</u>nabled	6. <u>A</u>borted
3. <u>X</u>ecuting	7. <u>T</u>imedOut
4. <u>C</u>ompleted	8. <u>S</u>kipped

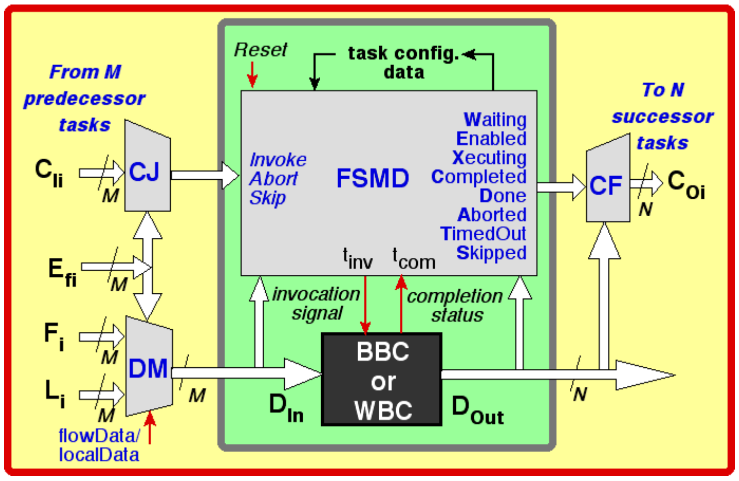
Whenever the FSMD receives a 'reset' pulse signal, it returns to the <u>W</u>aiting state. From this state, the FSMD transitions to the <u>E</u>nabled state upon receiving the 'invocation' pulse signal from the ControlJoin primitive task. From this state, the FSMD transitions to the <u>X</u>ecuting state, invoking the blackbox component, only upon receiving the 'host' pulse signal that the designated host is available. When in <u>X</u>ecuting state, the FSMD may transition to <u>W</u>aiting, <u>A</u>borted, <u>T</u>imedOut, or <u>C</u>ompleted state, depending whether it receives a 'reset', an 'abort', a 'timed-out', or a 'completed' pulse signal. When in a <u>C</u>ompleted state, the FMSD may transition either to <u>E</u>nabled state (and repeat invocation of the blackbox component if the host is available), or transition to <u>D</u>one state, if the 'done' pulse signal is generated by the datapath to terminate the repeated invocation of the task. A complete state-transition table and datapath table of the FSMD primitive task is available in [11].

We next expand on the representation of ControlJoin, DataMux, and ControlFork as introduced in Figure 3.

ControlJoin (Default Conditions). The ControlJoin primitive generates one of the three types of 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 (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.

In addition to the above default join condition, it is possible to specify more complex join conditions [11]. Some of these conditions are analyzed in Section 5.

DataMux. The DataMux 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_{Il_i} in Figure 3 (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 correspond-



The architecture of the task instance is based on the arrangement of the abstract task primitives FSMD, ControlJoin (CJ), DataMux (DM), ControlFork (CF) and BlackBox (WhiteBox) components introduced in Figure 2. FSMD has a total of eight states, outlined in the paper. A complete description of FSMD is presented in [11]. Functional descriptions of CJ, DM and CF are given below.

Repeated task invocation is controlled by FSMD. This feature is captured as *RepeatCondition* in the taskflow schema in Figure 4.

(a) ControlJoin conditions:

Default join:

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

For other representative join conditions, see [11].

(b) DataMux Table:

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

(c) ControlFork table:

Inputs								n^{th} **Outputs**							
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}
1	0	0	0	0	0	0	x	1	0	0	0	0	0	0	0
0	1	0	0	0	0	0	x	0	1	0	0	0	0	0	0
0	0	1	0	0	0	0	x	0	0	1	0	0	0	0	0
0	0	0	1	0	0	0	x	0	0	0	1	0	0	0	0
0	0	0	0	1	0	0	x	0	0	0	0	1	0	0	0
0	0	0	0	0	1	0	x	0	0	0	0	0	1	0	0
0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1	0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	†	0	0	0	0	0	0	1	0
0	0	0	0	0	0	1	‡	0	0	0	0	0	0	0	1

† 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: The architecture of the task instance.

ing control-edge E_{f_i}. Thus, the DataMux primitive selects the flow data only when E_g and E_{f_i} are both enabled.

ControlFork Table. The ControlFork primitive represents 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 (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.

4. TASKFLOW SCHEMA & SCHEDULING

In choosing the asynchronous FSM model to encapsulate each blackbox as well as each whitebox component, we have shown preference for the traditional (and simpler) approach, proven since 1950's in the design of increasingly complex hardware components and systems. Electronic circuit design in particular has a long tradition of addressing problems of concurrency and synchronization. The design of systems based on *interacting FSMs*, synchronous and asynchronous, is common. Alternative approaches, such as the formalisms of Petri nets [18], Actor Computations [19], Action Systems [20], Temporal Logic of Actions [21], appear less suitable – at present.

The instantiation of each encapsulated task with primitives such as ControlJoin, DataMux, and ControlFork bears similarities, in principle at least, to Hoare Logic and its extension with auxiliary variables [22]. However, the motivation for this paper, as clearly stated in Section 2, is (1) to formalize a basis for implementation of a universal client, and (2) to provide the user with a tool to interactively compose and execute new programs from existing component programs. The emphasis here is on '*clarity about the interface conditions between the building blocks out of which the*

The taskflow schema is based on the task instance architecture. The *MainTask* layer serves to invoke any TaskGraph of task instances. The *TaskInstance* layer is always an arrangement of an *EncapsulatedTask* with *ControlJoin*, *DataMux* and *ControlFork* primitives, and optionally, *RepeatCondition*.

The encapsulated task layer assigns, via *FSMD*, states to *blackbox* (*whitebox*) components. There are two sublayers: a *single-task definition* (*multi-task definition*) and a *single-task body* (*multi-task body*). The definition layer contains *I/O port* lists, and in case of a whitebox, a *TaskGraph* of task instances. This layer also serves as an *API* for the task.

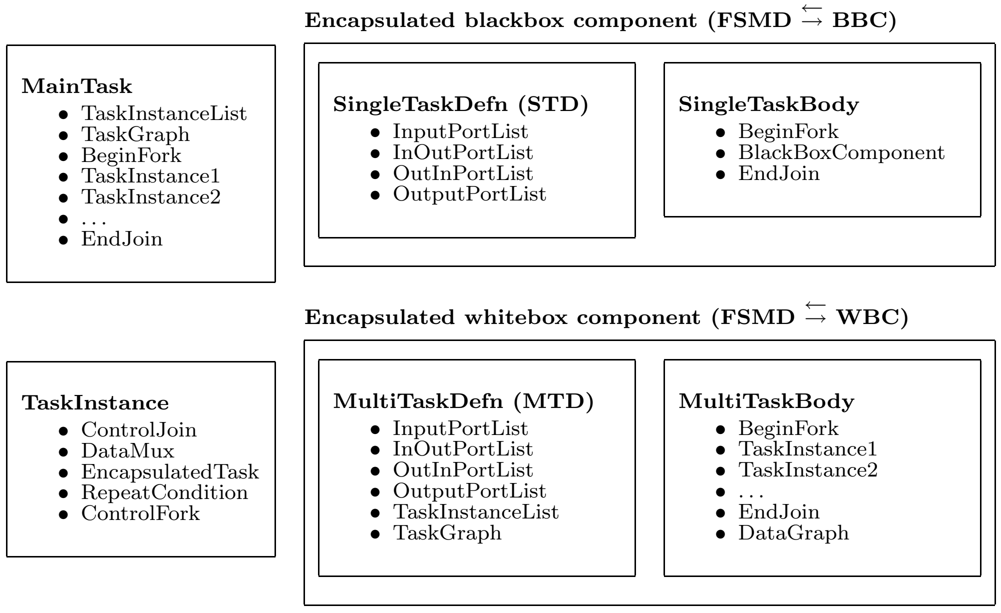


Figure 4: A schema to represent taskflow layers.

user composes his program' [15], not on program verification.

The main goals of this section are: (1) to present a taskflow schema based on the proposed taskflow architecture, and (2) to introduce a simple but effective taskflow scheduling algorithm.

4.1 Taskflow Schema

The schema to construct a taskflow consists of mainly two layers: an encapsulated blackbox (single-task) or whitebox (multi-task) layer, and a task instance layer. The schema for task instance layer is identical to the conceptual description in Section 3.2. However, the schema for the encapsulated task layer contains two distinct sublayers: a *single-task definition* (*multi-task definition*) layer, and a *single-task body* (*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 4 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, introduced in Section 2. 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.

The task body layer, as the name suggests, contains more detailed information about the task. 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 options to invoke tasks based on time-stamps of input/output data, presence or absence of certain data files, etc. As for the blackbox component, user needs to specify the details necessary to invoke the blackbox 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 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 is created by adding a *ControlJoin*, *DataMux* and a *ControlFork* elements, and optionally a *RepeatCondition*. 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 whitebox, 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 user-configurable 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 whitebox and the main task is that the latter does not have any flow data dependencies. Thus, each task instance depends on other task instances as determined by the the task graph which schedules the actual sequence of execution.

```
For each task $T_k$ $(k > 0)$ {
    /* evaluate ControlJoin conditions */
    $\{P_{I_k}, P_{A_k}, P_{S_k}\} \equiv \mathbf{CJ}_k(\mathcal{Q}_m, \mathcal{E}_{f_m}, \mathcal{E}_{a_p})$   $\forall m \in \mathrm{pred(T_k)},\ \forall \mathrm{p} \in \mathrm{abort(T_k)}$
        where
        $\mathcal{Q}_m = \{Q_{E_m}, Q_{X_m}, Q_{C_m}, Q_{T_m}, Q_{A_m}, Q_{S_m}, Q_{V_m}, Q_{N_m}\}$

        /* evaluate FSMD$_k$ conditions */
        if $P_{I_k} = 1$ 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$_k$: $t_{\mathrm{inv_k}} \equiv Q_{X_k} = t_{h_k}.E_k$

                        if $t_{\mathrm{com_k}} = 1$ (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 $t_{\mathrm{inv_k}}$ 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 $P_{I_k} = 1$.

        if $(P_{A_k} = 1)$ 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 $(P_{S_k} = 1$ 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 */
    $\mathcal{Q}_n \equiv \mathbf{CF}_k(\mathcal{Q}_k, \mathcal{D}_{O_k})$   $\forall n \in \mathrm{succ(T_k)}$
        where $\mathcal{D}_{O_k}$ is the set of output data values produced by task $T_k$, and
        $\mathcal{Q}_n = \{Q_{E_n}, Q_{X_n}, Q_{C_n}, Q_{T_n}, Q_{A_n}, Q_{S_n}, Q_{V_n}, Q_{N_n}\}$
        $\mathcal{Q}_k = \{Q_{W_k}, Q_{E_k}, Q_{X_k}, Q_{C_k}, Q_{T_k}, Q_{A_k}, Q_{S_k}, Q_{D_k}\}$
}
```

Figure 5: Outline of the scheduling algorithm for task T_k and its successors.

4.2 Taskflow Scheduling

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. Major evaluations take place within the ControlJoin, FSMD and ControlFork.

Let $m \in \mathrm{pred(T_k)}$ designate m control-edge predecessors of task T_k; $p \in \mathrm{abort(T_k)}$ designate p abort-edge incoming tasks of task T_k (these could include successors of T_k); and let $\mathbf{CJ}_k(\mathcal{Q}_m, \mathcal{E}_{f_m}, \mathcal{E}_{a_p})$ designate the assignments evaluated by the ControlJoin, where $\mathcal{Q}_m = \{Q_{E_m}, Q_{X_m}, Q_{C_m}, Q_{T_m}, Q_{A_m}, Q_{S_m}, Q_{V_m}, Q_{N_m}\}$ 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, $\forall m \in \mathrm{pred(T_k)},\ \forall \mathrm{p} \in \mathrm{abort(T_k)}$:

$$\{P_{I_k}, P_{A_k}, P_{S_k}\} \equiv \mathbf{CJ}_k(\mathcal{Q}_m, \mathcal{E}_{f_m}, \mathcal{E}_{a_p})$$

where $\{P_{I_k}, P_{A_k}, P_{S_k}\}$ 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

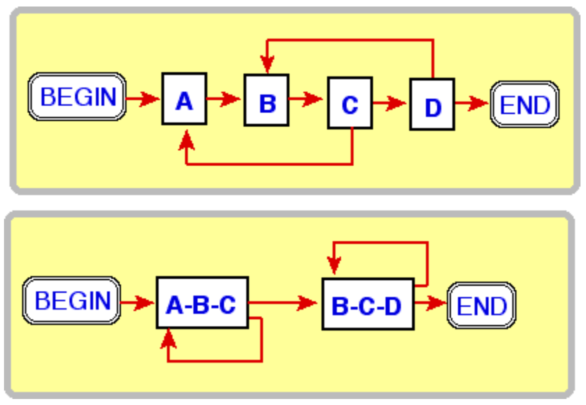
$$\mathcal{Q}_n \equiv \mathbf{CF}_k(\mathcal{Q}_k, \mathcal{D}_{O_k}) \quad \forall n \in \mathrm{succ(T_k)}$$

where $n \in \mathrm{succ(T_k)}$ designates n control-edge successor states of task T_k;
$\mathcal{Q}_n = \{Q_{E_n}, Q_{X_n}, Q_{C_n}, Q_{T_n}, Q_{A_n}, Q_{S_n}, Q_{V_n}, Q_{N_n}\}$;
$\mathcal{Q}_k = \{Q_{W_k}, Q_{E_k}, Q_{X_k}, Q_{C_k}, Q_{T_k}, Q_{A_k}, Q_{S_k}, Q_{D_k}\}$; and
$\mathcal{D}_{O_k}$ 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 FSMD [11]. The overall scheduling algorithm that takes the transition table and the Datapath table of the FSMD$_k$ into account is outlined in Figure 5. Nominally, the user will initiate the invocation of the BeginFork task, i.e. task for which $k = 0$, T_0. Tasks can be executed concurrently – but only 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 $\mathbf{CJ}_k$ will have been satisfied. The last task to be evaluated will always be the EndJoin task, since the TaskGraph is polar.

5. TASKFLOW PATTERN EVALUATIONS

Features of the taskflow environment can be evaluated in terms of the distinctive scheduling patterns: from simple sequencing of tasks, to splits, concurrency, joins, iterations, and cycles. A number of such patterns have been identified, evaluated and reported in the context of workflow environments in [8]. We use these patterns to evaluate the functionality of the environment created by the proposed OmniFlow



Arbitrary cycles such as shown with tasks A, B, C and D can result in a deadlock as well as infinite loops. Here, task A is never invoked because it waits for an invocation by task C indefinitely, 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 B-C-D-B, the loop A-B-C-A may start executing inadvertently.

We therefore allow only structured cycles, taking the form of a self-loop at the respective level of hierarchy. In this example, we transform the arbitrary cycles into structured cycles by replicating the instances of tasks B and C, and forming a hierarchical taskflow of two task chains with a self-loop in repeating the tasks in each of the two task chains: A-B-C and B-C-D.

Figure 6: Restructuring a pattern of arbitrary cycles to a pattern with structured cycles.

universal client. An example of the taskflow restructuring that removes the arbitrary taskflow cycles is illustrated in Figure 6. Additional comparisons are are summarized in Figure 7.

As reported in [8], the MQSeries/Workflow [9] supports a direct implementation of five out of ten patterns, whereas all other environments, also evaluated in [8], support less patterns. As for the OmniFlow environment, it supports, by default, seven out of ten patterns listed in the table. Also, the OmniFlow environment also supports additional patterns, not listed in Figure 7, such as recursion and manual synchronization. We now briefly discuss each of the patterns in the order listed in Figure 7:

Synchronizing merge: A synchronizing merge pattern involves two or more tasks that can be executed concurrently, followed by a single task, such as tasks *A1, A2, B1* followed by the task *C1* in Figure 1. There are a number of cases that should be considered when two or more tasks are executed concurrently. Consider an example of three tasks: A, B, and C, where tasks A and task B are executing concurrently and task C is the common successor task for both A and B. A rigid requirement may stipulate that the task C should not be invoked unless *all* of its predecessor tasks, here A and B, have completed execution. A more flexible scenario may allow task C to proceed when either task A or task B have been invoked and completed. In general, the synchronizing merge pattern in this example may represent the following three cases:

1. When both tasks A and B are available for execution, task C should wait for both A and B to complete execution.
2. When only one of the two tasks, A or B, is available for execution, task C should be invoked as soon as A or B completes execution since only one of the two would have been invoked.
3. When none of the the two tasks is available for execution, task C should not be invoked all.

A *logical AND-join* of tasks A and B is sufficient to resolve the first and the third case, but it fails for the second case because task B, which is not invoked, prevents the task C from executing even after task A completes. On the other hand, changing to a *logical OR-join* of tasks A and B resolves the second case, but fails for the first case.

The problem to invoke task C correctly is solved by specifying a *combination of AND/OR* join conditions using the *states* of tasks A and B. The join condition for the current example is thus: *OR(AND(A valid, B valid), AND(A valid, B skip), AND(A skip, B valid))*.

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. An example of such a pattern has been discussed in Section 2, Figure 1.

Arbitrary cycles: Most taskflows contain cycles because a sequence of tasks must be iterated several times to complete the overall task. However, arbitrary cycles pattern can potentially lead to infinite loops and deadlock conditions, unless designed properly. We can always transform a taskflow with arbitrary cycles into a taskflow containing structured cycles only, i.e. taskflows with self-loops such as shown in Figure 6.

Implicit termination: Implicit termination pattern occurs when taskflow has 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 OmniFlow environment, we prevent such problems by insisting on a single entry point and a single exit point. In the example above, tasks C and D should *both* be connected to an EndJoin primitive task.

Multiple 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 (self-loops) 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.

Workflow Pattern	**Level of Support** **Environments evaluated in [8]**	**OmniFlow Env.**
Synchronizing merge	MQSeries/Workflow and Inconcert support merge, others implement using X-OR split constructs	yes, `ControlJoin` construct directly supports merge
L-out-of-M join	Verve uses 1-out-of-M join (discriminator) combined with AND join and AND splits, hard to implement in others	yes, `ControlJoin` construct directly supports L-out-of-M join
Arbitrary cycles	MQSeries/Workflow, Inconcert has decomposition construct, Visual WorkFlo, SAP R/3 has special loop construct	partial, arbitrary cycles need to be transformed into structured cycles
Implicit termination	Staffware, MQSeries/Workflow, Inconcert terminate processes when idle, others allow only single exit node	allows only single exit node, so implicit termination does not occur
Multiple Instances (apriori knowledge)	MQSeries/Workflow provides special 'Bundle' construct to instantiate number of instances, others do it sequentially	yes, use structured cycles
Multiple Instances (no apriori knowledge)	Forte, Verve use loop and parallel split, Visual WorkFlo supports Release, I-Flow supports Chained Process Node	yes, use structured cycles
Multiple Instances requiring synchronization	MQSeries/Workflow's 'Bundle' construct allows synchronization of created instances, not easy to implement in others	partial, limited to simple join conditions
Deferred choice	cancel other choice when selected for execution, or add an extra task to implement implicit-OR using explicit-OR construct	yes, `ControlJoin` construct directly supports deferred choice
Interleaved routing	pre-define sequence, or use deferred XOR-split construct, or for petri-net based models, add extra place as input/output of concurrent activities	no, this pattern is not directly supported
Milestone	introduce a boolean variable and check its value periodically	yes, use `ControlJoin` construct
Overall	MQSeries/Workflow supports a direct implementation of 5/10 patterns, all others support less	OmniFlow environment supports a direct implementation of 7/10 patterns.

Figure 7: Summary of ten advanced workflow patterns.

Multiple instances with no apriori runtime knowledge: In addition to invoking task instances dynamically during runtime, it may be possible that the number of instances of task invocation is not known prior to task execution. This can happen when the task instances depend on one another. For this pattern, we condition the data-dependent task repetitions that iterate a single task a number of times until the required condition on data set is satisfied.

Multiple instances requiring apriori 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 the OmniFlow environment, we only allow simple join conditions such as wait for L out of n task instances to complete.

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.

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

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 on-line store. The shipment of a purchased item can be expedited only as long as long as the item has not been shipped. Let task A represent the purchase order for an item, task B represent the request for expediting the shipment, task C represent the shipment of an item, and task D represent the processing the expediting request.

This problem is resolved by use of the *NOT* function in specification of the ControlJoin condition of task D. This condition prevents task D from executing if task C has already completed execution.

6. SUMMARY AND CONCLUSIONS

This paper introduces concepts of *a taskflow-oriented programming paradigm*, with task instances representing distributed stand-alone component programs, such that *users*, without assistance from expert programmers, can compose interactive, executable programs using these components. Rather than engaging a programmer to write a custom script each time a composition of a new software system is needed, *user alone* can now (1) write a *hierarchical* taskflow configuration, (2) invoke the *universal client (OmniFlow)* that reads the configuration and renders it as a highly interactive GUI, and (3) interact with the taskflow in a number of ways.

An XML/TclTk implementation of such a universal, user-configurable client is presented in the companion paper [10]. The recursive schema of component instances is conveniently captured as an extension of XML in a collaborative distributed task mark-up language (cdtML) and consists of mainly two layers: an encapsulated blackbox (single-task) or a whitebox (multi-task) layer, and a task instance layer. A generic Tcl-XML parser reads both the cdtML schema and the user-created cdtML taskflow description and outputs a taskflow description in TclTk. This in turn generates the interactive GUI as the hierarchical taskflow, waiting for user inputs. User may choose to interact in any of the following ways: reconfigure the taskflow interconnections, view or edit data, descend/ascend the taskflow hierarchy, select the mode of execution, invoke the taskflow, abort the taskflow (if already executing), reset the state of the taskflow, etc.

Acknowledgments. We thank Dr. Mladen Vouk, Dr. Munindar Singh, and Dr. Matthias Stallmann, all from NCSU, for the encouragement and the constructive feedback on formalizing the concepts of taskflow-oriented programming paradigm.

7. REFERENCES

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

[2] C. Szyperski and C. Pfister. Workshop on Component-Oriented Programming, Summary. Special Issues in Objected Oriented Programming - ECOOP, 1997.

[3] Object Management Group Home Page, May 2001. See `http://www.omg.org/`.

[4] Java Technology Home Page, May 2001. See `http://www.javasoft.com/`.

[5] Component Object Model (COM) Technologies Home Page, May 2001. See `http://www.microsoft.com/com/`.

[6] C. Wiederhold and P. Wegner and S. Ceri. Toward Megaprogramming. *Communication of ACM*, 35(11):89–99, 1992.

[7] J. K. Ousterhout. Scripting: Higher Level Programming for the 21st Century. Article in *IEEE Computer* Magazine, March 1998. Also available at `http://scriptics.com/people/john.ousterhout/-scripting.html`.

[8] W.M.P. van der Aalst and A.P. Barros and A.H.M. ter Hofstede and B. Kiepuszewski. Advanced Workflow Patterns. Proceedings Seventh IFCIS International Conference on Cooperative Information Systems, September 2000. Also available at `http://www.icis.qut.edu.au/~arthur/articles/-apatterns.pdf`.

[9] MQSeries Adaptive MiddleWare , May 2001. See `http://www-4.ibm.com/software/ts/mqseries/`.

[10] H. Lavana and F. Brglez. A Universal Client for Taskflow-Oriented Programming with Distributed Components: an XML/TclTk Implementation. In *The 8th Tcl/Tk Conference at the O'Reilly Open Source Convention*. O'Reilly, July 2001. See also `http://www.cbl.ncsu.edu/publications/-#2001-TclTk-Lavana`.

[11] H. Lavana. *A Universally Configurable Architecture for Taskflow-Oriented Design of a Distributed Collaborative Computing Environment*. PhD thesis, Electrical and Computer Engineering, North Carolina State University, Raleigh, N.C., December 2000. Also available at `http://www.cbl.ncsu.edu/-publications/#2000-Thesis-PhD-Lavana`.

[12] F. Brglez, H. Lavana, D. Ghosh, B. Allen, R. Casstevens, J. Harlow III, R. Kurve, S. Page, and M. Stallmann. OpenExperiment: A Configurable Environment for Collaborative Experimental Design and Execution on the Internet . Technical Report 2000-TR@CBL-02-Brglez, CBL, CS Dept., NCSU, Box 8206, Raleigh, NC 27695, March 2000.

[13] 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. IEEE Intl. Conference on Computer Design, 2000. Also available at `http://www.cbl.ncsu.edu/-publications/#2000-ICCD-Lavana`.

[14] N. Wirth. On the Composition of Well-Structured Programs. *Computing Surveys*, 6(4):274–259, December 1974.

[15] P. Naur. Proof of Algorithms by General Snapshots. *BIT*, 6(4):310–316, 1966.

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

[17] G. DeMicheli. *Synthesis and Optimization of Digital Circuits*. McGraw Hill, 1994.

[18] W.M.P. van der Aalst and A.H.M. ter Hofstede. Verification of Workflow Task Structures: A Petri-net-based approach. Information Systems Journal, March 2000.

[19] G. A. Agha and I. A. Mason and S. Smith and C. Talcott. A Foundation for Actor Computation. Journal of Functional Programming, 1996. Also available at `http://osl.cs.uiuc.edu/Papers/-actor-semantics.ps`.

[20] R. J. R. Back and Kuri-Suonio . Distributed Co-operation with action systems. Transactions on Program Languages and Systems, 1988.

[21] L. Lamport. Temporal Logic of Actions Home Page, May 2001. See `http://research.compaq.com/SRC/-personal/lamport/tla/tla.html`.

[22] T. Kleymann. Hoare Logic and Auxiliary Variables. *Formal Aspects of Computing*, 11(5):541–566, 1999.