A Universal Client for Taskflow-Oriented Programming with Distributed Components: an XML/TclTk Implementation [*]

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

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

ABSTRACT

This paper presents an XML/TclTk implementation of a universal, user-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 client creates a taskflow-oriented programming environment, conceptually introduced in the companion paper and demonstrated in this paper.

The recursive schema of component instances is conveniently captured as an extension of XML in a collaborative distributed taskflow mark-up language (cdtML) and consists of mainly two layers: (1) an encapsulated blackbox (single-task) or a whitebox (multi-task) layer, and (2) a task instance layer. The encapsulated task layer contains two parts: a definition layer and a body layer, with the definition layer serving as a readily accessible API for the task. 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.

1. INTRODUCTION

The concepts of taskflow-oriented programming with distributed components, introduced in [1] and the focus of the companion paper [2], provide the basis for the implementation of a universal, user-configurable client presented in this paper. In principle, the client may be implemented in any number of popular programming languages – and the user may not particularly care about this choice. What the user does care about is the choice of the language in which he is to specify the interconnection and the interfaces of the distributed components.

There are three main reasons why we represent the interconnections and the interfaces of the distributed components as an extension of XML in a collaborative distributed taskflow mark-up language (cdtML): (1) a *cdtML schema* can be readily created using XML [3, 4], (2) an increasing number of XML parsers and validating XML editors are available on the Internet [5], (3) user acceptance and familiarity with XML will continue to grow with time.

Based on our own experience, there are a number of reasons to choose TclTk [6] as the programming language to implement the universal client (OmniFlow), including: (1) prior implementation experiences [7, 8, 9], (2) extensions to render the TclTk-created GUI in the web-browser [10, 11], (3) extensions to render the TclTk-created GUI collaborative [12], (4) availability of an XML-Tcl parser [13], (5) availability of general purpose TclTk resources, e.g. [14], and (6) last but not least, the ability to readily write TclTk applications such that they are cross-platform compatible by design. If we do not access components via telnet or ssh (when the implementation also requires *expect* [15]), the OmniFlow client described in this paper executes without modifications under unix, linux, MacOS, and WindowsNT operating systems.

Implementation Architecture. As described in [1, 2], 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 is an arrangement of exactly 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 associated with 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.

Such a schema is central to the architecture that implements the taskflow GUI, scheduling, and execution; both the schema and the architecture are outlined in Figure 1.

[*]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.

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.

MainTask
- TaskInstanceList
- TaskGraph
- BeginFork
- TaskInstance1
- TaskInstance2
- . . .
- EndJoin

TaskInstance
- ControlJoin
- DataMux
- EncapsulatedTask
- RepeatCondition
- ControlFork

Encapsulated blackbox component (FSMD $\leftrightarrows$ BBC)

SingleTaskDefn (STD)
- InputPortList
- InOutPortList
- OutInPortList
- OutputPortList

SingleTaskBody
- BeginFork
- BlackBoxComponent
- EndJoin

Encapsulated whitebox component (FSMD $\leftrightarrows$ WBC)

MultiTaskDefn (MTD)
- InputPortList
- InOutPortList
- OutInPortList
- OutputPortList
- TaskInstanceList
- TaskGraph

MultiTaskBody
- BeginFork
- TaskInstance1
- TaskInstance2
- . . .
- EndJoin
- DataGraph

The generic taskflow schema above has been implemented as an extension of XML (cdtML). The implementation of the taskflow GUI and its execution engine is divided into two parts: (1) initial loading of the generic cdtML schema and user-configured taskflow files in cdtML, and (2) scheduling task instances for execution during runtime. The taskflow loader parses taskflow configuration in cdtML, verifies the taskflow syntax against the cdtML schema and outputs the taskflow configurations as tcl files, loads the tcl taskflow configurations and renders an interactive GUI of the taskflow *tree and graph views*.

In the initial phase, only the top-level configuration is loaded and rendered as GUI; subsequent loading, rendering, scheduling, and executing is dynamic as described in the paper.

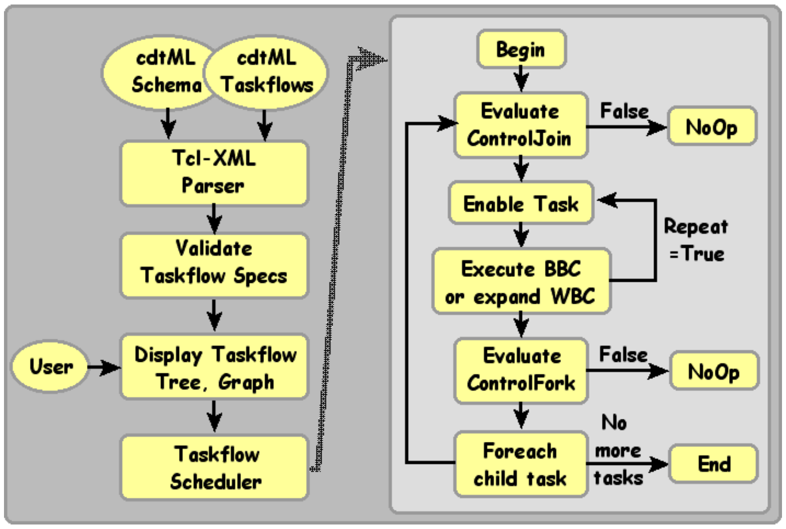


Figure 1: Implementation architecture of the taskflow GUI, scheduling, and execution engine.

While the schema, as introduced in [1, 2], does not imply a specific implementation, such a schema can also be readily implemented in XML and is presented in subsequent sections, including examples of taskflow descriptions in cdtML.

The implementation of the taskflow GUI and its execution engine is divided into two parts: (1) initial loading of the generic cdtML schema and user-configured taskflow files in cdtML, and (2) scheduling task instances for execution during runtime. The taskflow loader must (1) parse taskflow configuration specified in cdtML, (2) verify the taskflow syntax against the cdtML schema and output the taskflow configurations as Tcl files[1], (3) load the Tcl taskflow configurations and render a interactive GUI of the taskflow *tree and graph views*. In the initial phase, only the top-level configuration is loaded and rendered as GUI; subsequent loading, rendering, scheduling, and executing is dynamic and subject to selective user-interactions with the taskflow GUI.

The recursive algorithm that implements the taskflow scheduler calls itself whenever it encounters a whitebox instance. Initially, all instances are in a *Waiting* state. When a task instance is invoked, the taskflow scheduler first evaluates its *ControlJoin* and accordingly changes the state of the task to *Enabled* state. Once enabled, and if the instance is a whitebox, it immediately changes to *Executing* state. The scheduler expands the whitebox instance into another level of hierarchy and schedules the *(BEGIN)* node for execution. On the other hand, if it the instance is a blackbox, it is scheduled for execution only after the designated host becomes idle.

Upon completion of first task execution, scheduler evaluates the *RepeatCondition* for the task instance, if any, to re-invoke the task until the condition is satisfied. After that,

[1]The current Tcl-XML parser [13] does not yet support the validation against an XML schema.

it evaluates the *ControlFork* to validate the status of each of its output InvocationEdges. The process repeats for each of the successor task of the task instance.

Illustrative Example. Throughout the paper, we use segments of the taskflow example *'parabola'*. In Figure 2, we illustrate the two key phases of taskflow-oriented programming using OmniFlow: (1) an entry of a representative taskflow description in cdtML, and (2) a rendering of the taskflow GUI in a state determined by the user interactivity.

We first note how the entry of the taskflow description is facilitated by using a validating XML editor, i.e. an editor that not only reads the cdtML schema in XML but also uses it to validate user entries of the cdtML taskflow descriptions[2]. In the example shown, the editor has been configured by the cdtML schema in XML to validate elements of the generic multi-task definition schema introduced in Figure 1 such as the InputList, the OutputList, the TaskList, and the TaskGraph, and to support user entries of specific instances of I/O ports, task instances, and task dependencies.

Once the cdtML descriptions of the related taskflows are completed, a 'main' taskflow instance is read by the OmniFlow client, which in turn parses the cdtML description, writes a corresponding description in TclTk, and loads this description to render the top-level of the taskflow GUI. At this point, it is up to the user to interact with the taskflow: to invoke it at the top level or to descend the hierarchy and invoke one or more taskflows at any of the lower levels. Only at this point are the additional cdtML descriptions parsed, converted to TclTk, and loaded for rendering as new taskflows by the OmniFlow client. In Figure 2, the taskflow *parabola_main.cdt* is linked to a number of underlying cdtML taskflow descriptions. Upon invocation of *parabola_main.cdt* and some user interaction with its initial rendering, user can expand a representation of the entire taskflow, including the taskflow instance (C) as shown.

The rendering of the taskflow in Figure 2 represents a hierarchical *graph view* of the taskflow, in contrast to the hierarchical *tree view* that is implicitly represented by the taskflow schema and the taskflow capture in cdtML. Clearly, the graph view significantly complements the more traditional tree view. For example, the *invocation edges* from task instance ($BEGIN$) to task instances ($InitA$), ($InitB$), ($InitC$), signify the possibility of concurrent invocation and execution of these tasks. Similarly, the *repeat invocation edges* associated with the hierarchical task instances (A), (B), and (C) signify a repeated invocation of these tasks until some terminating condition is satisfied. An entire section in the paper is devoted to presenting details about these views and about user interaction that can reconfigure the taskflow and its scheduled execution dynamically.

Paper Organization. As illustrated in the preceding examples, there are two main components that we bring together in the implementation of the universal client as defined in the companion paper [2]: (1) parsing taskflow cdtML descriptions in accordance with a well-defined schema in XML and creating executable TclTk programs, and (2) rendering an interactive GUI such that the hierarchy of of taskflow instances and the respective GUIs can be invoked, scheduled, and executed dynamically in response to user interactions. The next two sections, (2) Taskflow Schema and User Descriptions in cdtML, and (3) Taskflow GUI and Interactions, describe the proposed approach in some detail. The effectiveness of the implementation is subject of Performance Experiments in Section 4. The paper concludes with Section 5, Summary and Conclusions.

[2]The screenshot of the validating XML editor in Figure 2 is based on the demo version of the XMLSpy editor [16].

2. TASKFLOW SCHEMA AND TASKFLOW DESCRIPTIONS IN CDTML

The taskflow schema, illustrated in Figure 1 and described in detail in [2], supports a specific task instance architecture but does not imply a particular implementation language. Our choice of XML to implement this schema as an extension of XML in a collaborative distributed taskflow mark-up language (cdtML) has been articulated in Section 1. In this section, we describe

- a cdtML taskflow schema using XML, and
- taskflow descriptions in cdtML.

To illustrate representative taskflow descriptions, we will continue to use segments of the taskflow example *'parabola'*, introduced in Figure 2.

2.1 A cdtML Taskflow Schema Using XML

As illustrated in Figure 1, 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 the encapsulated task layer contains two distinct layers: (1) a blackbox or whitebox *definition* layer, and (2) a blackbox or whitebox *body* layer. This distinction helps in separating the task API from its body declaration, which can be very detailed.

Consider elements of the whitebox definition layer (*MultiTaskDefn*) in Figure 1, starting with *InputPortList* and ending with *TaskGraph*. In Figure 3, we show a corresponding cdtML schema, implemented in XML. The whitebox definition layer is now declared as an element of *complex type* with name *MultiTaskDefn*. This type is shown as a sequence of seven elements within it, including *Title*, *Description*, *Input-*

```
<!-- type declaration of whitebox (multi-task) definition -->

<xsd:complexType name="MultiTaskDefn" content="elementOnly">
  <xsd:sequence minOccurs="0" maxOccurs="1">
    <xsd:element name="Title"       type="Title"/>
    <xsd:element name="Description" type="Description"/>
    <xsd:element name="InputList"   type="InputList"/>
    <xsd:element name="InOutList"   type="InOutList"/>
    <xsd:element name="OutInList"   type="OutInList"/>
    <xsd:element name="OutputList"  type="OutputList"/>
    <xsd:element name="TaskList"    type="TaskList"/>
    <xsd:element name="TaskGraph">
      <xsd:complexType content="textOnly">
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
  <xsd:attribute name="name">
    <xsd:simpleType base="xsd:string"/>
  </xsd:attribute>
</xsd:complexType>
```

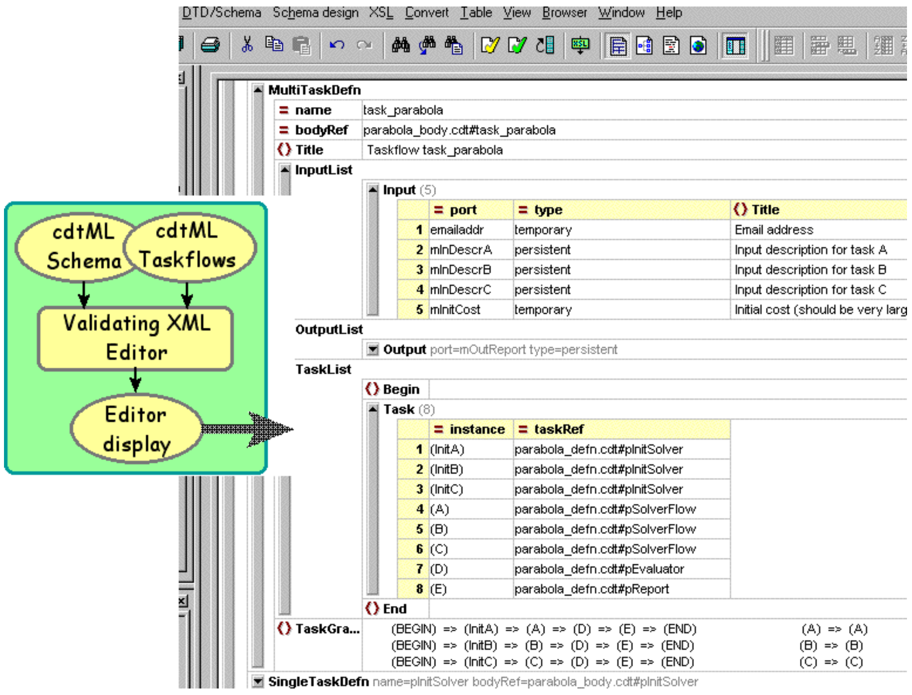
Figure 3: A section of cdtML schema using XML.

```
 **************   cdtml_schema.xml   ***************

<!-- type declaration of whitebox (multi-task) definition -->
<xsd:complexType name="MultiTaskDefn" content="elementOnly">
  <xsd:sequence minOccurs="0" maxOccurs="1">
    <xsd:element name="Title"       type="Title"/>
    <xsd:element name="Description" type="Description"/>
    <xsd:element name="InputList"   type="InputList"/>
    <xsd:element name="InOutList"   type="InOutList"/>
    <xsd:element name="OutInList"   type="OutInList"/>
    <xsd:element name="OutputList"  type="OutputList"/>
    <xsd:element name="TaskList"    type="TaskList"/>
    <xsd:element name="TaskGraph">
      <xsd:complexType content="textOnly">
      </xsd:complexType>
    </xsd:element>
  </xsd:sequence>
.................
```

```
 **************   parabola_main.cdt   ***************

<!-- parabola_main.cdt -->
<Cdtml>
<MainTask>
    <TaskList>
        <Begin/>
        <Task instance="(parabola)"
               taskRef="parabola_defn.cdt#parabola"/>
        <End/>
    </TaskList>
    <TaskGraph>
        (BEGIN)  =>  (parabola)  =>  (END)
    </TaskGraph>
    <TaskInstance instance="(parabola)">
        <SetInput port="emailaddr">
            <LocalValue> lavana@cbl.ncsu.edu </LocalValue>
.................
```

While a cdtML description of a taskflow such as *parabola_main.cdt* above right can be entered directly, the preferred choice is to enter the description via an XML editor that also reads the XML description of the *cdtML schema* above left and can thus *validate* the syntax of the user taskflow description in cdtML.

Note how the structure of the user-instantiated schema displayed in the editor tracks closely the structure of the generic multi-task definition schema described in Figure 1. User entries, from the InputList, the OutputList, the TaskList, and the TaskGraph, include specific instances of I/O ports, task instances, and task dependencies.


cdtML Schema
cdtML Taskflows
Validating XML Editor
Editor display
DTD/Schema Schema design XSL Convert Table View Browser Window Help

MultiTaskDefn

= name	task_parabola
= bodyRef	parabola_body.cdt#task_parabola
() Title	Taskflow task_parabola

InputList

Input (5)

	= port	= type	() Title
1	emailaddr	temporary	Email address
2	mInDescrA	persistent	Input description for task A
3	mInDescrB	persistent	Input description for task B
4	mInDescrC	persistent	Input description for task C
5	mInitCost	temporary	Initial cost (should be very larg

OutputList

Output port=mOutReport type=persistent

TaskList

() Begin

Task (8)

	= instance	= taskRef
1	(InitA)	parabola_defn.cdt#pInitSolver
2	(InitB)	parabola_defn.cdt#pInitSolver
3	(InitC)	parabola_defn.cdt#pInitSolver
4	(A)	parabola_defn.cdt#pSolverFlow
5	(B)	parabola_defn.cdt#pSolverFlow
6	(C)	parabola_defn.cdt#pSolverFlow
7	(D)	parabola_defn.cdt#pEvaluator
8	(E)	parabola_defn.cdt#pReport

() End

```
() TaskGra...   (BEGIN) => (InitA) => (A) => (D) => (E) => (END)        (A) => (A)
                (BEGIN) => (InitB) => (B) => (D) => (E) => (END)        (B) => (B)
                (BEGIN) => (InitC) => (C) => (D) => (E) => (END)        (C) => (C)
```

SingleTaskDefn name=pInitSolver bodyRef=parabola_body.cdt#pInitSolver

Taskflow description of *parabola_main.cdt* above right is linked to a number of underlying cdtML taskflow descriptions. Upon invocation of *parabola_main.cdt* and some user interaction with its initial rendering, user can expand a representation of the entire taskflow, including the taskflow instance (C) as shown.

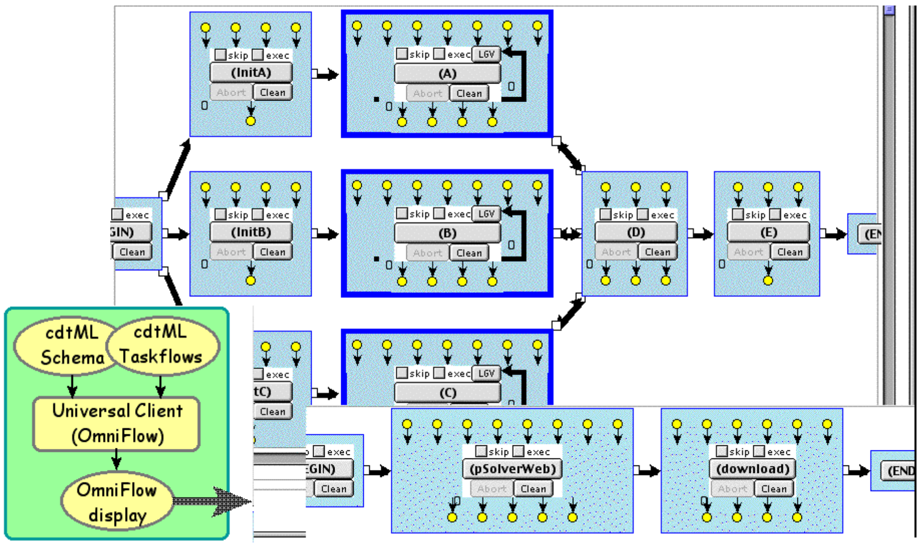


Figure 2: Taskflow configuration entry in a validating XML editor and its rendering via OmniFlow.

List, *InOutList*, *OutInList*, *OutputList*, *TaskList* and *TaskGraph*. However, each of these elements is optional and is specified using the 'minOccurs' and 'maxOccurs' attributes for the 'xsd:sequence' tag. The first six elements are of type defined elsewhere in the schema specification, whereas the seventh element, *TaskGraph*, is of type *textOnly*. The TaskGraph is described with task instance names as nodes, where the nodes are connected by up to three types of directed control edges: *InvocationEdge*, *RepeatInvocationEdge*, and *AbortInvocationEdge*. Full details about task graph representation are given in [2], example descriptions are shown in Figure 5, and discussed later.

In addition to these elements, the *MultiTaskDefn* layer has an attribute 'name' which consists of a string of type *simpleType*. This name is used to refer to the *MultiTaskDefn*.

A complete cdtML taskflow schema in XML is listed in the Appendix of [1]. For more details about XML, see [3].

2.2 Taskflow Descriptions In cdtML

In Figure 2 we used segments of the taskflow *'parabola'* to illustrate the two key phases of taskflow-oriented programming using OmniFlow client: (1) an entry of a representative taskflow description in cdtML, and (2) a rendering of the taskflow GUI in a state determined by the user interactivity. In this section, we give two views of the 'parabola' example: (1) as a case of project decomposition into tasks that are to be implemented by distributed components, and (2) as a case of structured taskflow descriptions using cdtML. In the section that follows, we use the same example to describe views of the taskflow GUI and a range of user interactions with it.

Parabola – Project Decomposition. This project has been devised to quickly illustrate (1) main features of distributed component-based computation, (2) concurrent execution of some such components, (3) synchronization issues that arise when executing components concurrently. Its structure is similar to the realistic example in Figure 1 of [2], however the implementation of each component is much simpler. We have tested three distinct taskflow implementations: (1) with all components residing on the same host (with some components executing concurrently), (2) with distributed components, accessed by way of telnet/ssh-based clients, and (3) with distributed components accessed by way of http-based clients [1]. In this paper, we present the 'parabola' taskflow where some components are accessed by way of a http-based client.

Consider the goal of computing an approximate minimum of a simple quadratic function by an iterative method. The approximation method we use is crude intentionally so that *both*, the number of iterations and the approximate minimum reported by the solver, depend critically on the starting point. Rather than analyze the problem by serially executing the solver from different starting points, we perform the analysis by executing multiple instances of the solver concurrently. The blackbox components at our disposal are:

(1) `pInitSolver`, a program on a local host. It initializes all data required by subsequent programs.

(2) `pSolverWeb`, a program accessible as a cgi script on a host specified by a URL. It finds an approximate minimum of a single variable function. It can be invoked repeatedly, using the solution from the previous iteration as a new starting point. User can 'control' the number of expected iterations before finding an approximate minimum by choosing a starting point far from the optimum value.

(3) `download`, a program on a local host. It downloads data from a directory on another host, referenced by a URL.

(4) `pEvaluator`, a program on a local host. The program finds the average of the n solutions and related statistics.

(5) `pReport`, a program on a local host. The program writes the solution report to a file.

Given these five blackbox components, we now aggregate them into whitebox (multi-task) component instances such that the computational tasks can be invoked and executed in the desired sequence. What we want is to invoke three instances of `pSolverWeb`, and in turn, also invoke three instances of `pInitSolver` and `download`. The hierarchical organization of the computational task instances we propose is shown in the *TaskTree* of Figure 4. The instances of `pInitSolver` are assigned names *(InitA)*, *(InitB)*, and *(InitC)*. The new whitebox component `pSolverFlow` is a composition two instances: *(pSolverWeb)* and *(download)* and we use three instances of this component, (A), (B), and (C). Here we also introduce the convention where '()' serve as the first and the last character of each task instance; this

```
TaskTree (parabolaMain)
----------------------
+ (parabolaMain)
  + (parabola)            # parabola      (whitebox)
     -  (InitA)           # pInitSolver   (blackbox)
     +  (A)               # pSolverFlow   (whitebox)
         -  (pSolverWeb)  # pSolverWeb    (blackbox)
         -  (download)    # download      (blackbox)
     -  (InitB)           # pInitSolver   (blackbox)
     +  (B)               # pSolverFlow   (whitebox)
         -  (pSolverWeb)  # pSolverWeb    (blackbox)
         -  (download)    # download      (blackbox)
     -  (InitC)           # pInitSolver   (blackbox)
     +  (C)               # pSolverFlow   (whitebox)
         -  (pSolverWeb)  # pSolverWeb    (blackbox)
         -  (download)    # download      (blackbox)
     -  (D)               # pEvaluator    (blackbox)
     -  (E)               # pReport       (blackbox)


TaskGraph (parabolaMain)
-----------------------
     (BEGIN) => (parabola) => (END)

TaskGraph (parabola)
--------------------
     (BEGIN) => (InitA) => (A) => (D) => (E) => (END)
                   (A) => (A)    (D) +> (A)
     (BEGIN) => (InitB) => (B) => (D) => (E) => (END)
                   (B) => (B)    (D) +> (B)
     (BEGIN) => (InitC) => (C) => (D) => (E) => (END)
                   (C) => (C)    (D) +> (C)

TaskGraph (A) or (B) or (C)
--------------------------
     (BEGIN) => (pSolverWeb) => (download) => (END)
```

Figure 4: Conceptual composition of 'parabola'.

convention is also an integral part of the more formal cdtML descriptions presented in the next subsection. There are a total of 15 task instances of single and multi-tasks defined under the root instance (parabolaMain).

However, the tree organization of tasks gives no information as to how any of these tasks *may* be scheduled for invocation. To complement the view of the relations between the task instances, we also introduce a *TaskGraph* at each level of the tree hierarchy. Examples of three such graphs are also shown in Figure 4. In each taskgraph we connect task instances with directed edges that can be of three types: InvocationEdge, RepeatInvocationEdge, and AbortInvocationEdge. Since the RepeatInvocationEdge can only imply a task instance with a self-loop, the symbols for the InvocationEdge and RepeatInvocationEdge are the same: `=>` represents the 'closed state' and `==` represents the 'open state' of the edge. We rely on the context of use to decide whether the edge is an InvocationEdge or a RepeatInvocationEdge. The symbols for the AbortInvocationEdge are `+>` for the 'closed state' and `++` for the 'open state'. In the taskgraph *(parabola)*, task instances (A), (B), and (C) have self-loops, and each one of them may potentially be aborted by a *valid* pulse on the AbortInvocationEdge from task (D). It is relatively easy to parse the directed graph descriptions as shown in Figure 4; the syntax we use is a derivative of the 'dot' syntax introduced in [17].

The user-defined taskgraph *(parabola)* implies a number of sequences in which task instances *may* be executed. If we suppress the AbortInvocationEdges, the task graph is always a polar DAG[3]. When analyzing a polar DAG in terms of its potential schedule, we may assume three defaults:

- a single task may invoke a number of tasks concurrently, given that the states of the outgoing InvocationEdges are *valid*.
- a single task, driven by a number of InvocationEdges is *not* invoked until the states of *all* incoming InvocationEdges are *valid*.
- given the RepeatInvocationEdge in a *valid* state, a single task may repeatedly re-invoke itself after the first invocation by another task.

The taskgraph of the (parabola) instance, as conceptually defined in Figure 4, is also part of the cdtML description in Figure 5. This description is read by the OmniFlow client which renders its tree and graph views as shown in Figure 6. Clearly, task instances *(InitA)*, *(InitB)*, and *(InitC)*, *may* be invoked concurrently, and so may task instances (A), (B), and (C) that follow. The BeginFork primitive within the task instance (D) has to decide the status of the three incoming InvocationEdges before invoking the instance (D) itself. If the task (D) is to be invoked after only two of the incoming InvocationEdges are *valid*, BeginFork must not only invoke task (D) but also assign a *valid* abort pulse to one of the Invocation AbortEdges incident at tasks (A), (B), or (C) that still may be executing. It is up to the user to refine the description of each task instance which in turn, will induce such a schedule, different from the defaults described above. Clearly, such schedule will also follow the directives implied by the user-placed AbortInvocationEdges.

[3]DAG stands for an abbreviation of a directed acyclic graph. A graph is polar since the task instances *(BEGIN)* and *(END)* are always present. Nodes in such a graph may also contain self-loops, indicating a repeating task, subject to some termination conditions.

Parabola – Taskflow cdtML Description. We continue to use the taskflow 'parabola' as the example to illustrate representative aspects of the cdtML syntax. Ideally, each cdtML description of a taskflow has been validated against the cdtML schema such as the XML segment shown in Figure 3. While not mandatory, the recommended file organization of a taskflow description for any major project is to divide the description into three files that capture the 'project main', 'project definition', and 'project body'.

As we illustrate in the next section, it is sufficient that only the 'project main' and 'project definition' files are available; OmniFlow client can render the entire hierarchy of the taskflow project on basis of these two files only. Moreover, user can interact with edges and execute any segment of the taskflow in the 'simulation mode', also described in the next section. This gives us the benefit of adjusting the composition and the sequencing of tasks *before* we commit to encapsulation of particular blackboxes. It also allows us to postpone the creation of any DataGraphs, i.e. specific assignments of data to task instances.

Now, proceeding with the cdtML description of the taskflow 'parabola', we illustrate the taskflow in three files:

- parabola_main.cdt (shown in its entirety in Figure 5),
- parabola_defn.cdt (two representative segments are shown in Figure 5, more in the Appendix, Figure 7),
- parabola_body.cdt (several representative segments are shown in the Appendix, Figure 8).

Due to space limitation, we present only selected highlights of cdtML descriptions. For more details, see the Appendix in [1] and the updated version in [18].

`parabola_main.cdt` in Figure 5 is a cdtML example of a *MainTask* layer introduced in Figure 1. The instance of *(parabola)* is referenced as "parabola_defn.cdt#parabola", i.e. pointing to its anchor in the file parabola_defn.cdt. Variable value 'lavana@cbl.ncsu.edu', file names 'pbInDescrA.d',..., 'parabola.rpt' are entered as data in this file.

`parabola_defn.cdt` in Figures 5 and 7 contain a number of cdtML examples of *Single (Multi) -Task Definition* layers introduced in Figure 1. Specifically, parabola represents a multi-task definition layer, supported as an element of *complex type* with name *MultiTaskDefn* in the cdtML schema in Figure 3. The correspondence between the elements of the cdtML schema and the user-entered description of the multi-task definition can be readily ascertained. Note also that the task graph, described conceptually in Figure 4 is now a formal part of the task graph description delimited by `<TaskGraph>` and `</TaskGraph>`.

As mentioned earlier, the descriptions of the MainTask and all TaskDefinition layers is sufficient to invoke the OmniFlow client and render the entire hierarchy of the taskflow project. When we are ready to include descriptions of corresponding TaskBody layer for each of the defined tasks, we reference the corresponding body description as follows:

```
<MultiTaskDefn name="parabola"
          bodyRef="parabola_body.cdt#parabola">
```

`parabola_body.cdt` in Figure 8 contains a number of cdtML examples of *Single (Multi) -Task Body* and *TaskInstance* layers introduced in Figure 1. These two layers are clearly the

```
**************   parabola_defn.cdt   ***************

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CDTML SYSTEM
            "http://www.cbl.ncsu.edu/OpenCdt/cdtml_schema.xml">
<!-- parabola_defn.cdt -->
<Cdtml>
<MultiTaskDefn name="parabola"
            bodyRef="parabola_body.cdt#parabola">
    <Title> Taskflow parabola </Title>
    <InputList>
        <Input port="emailaddr" type="temporary">
            <Title> Email address </Title>
        </Input>
        <Input port="mInDescrA" type="persistent">
            <Title> Input description for task A </Title>
        </Input>
        <Input port="mInDescrB" type="persistent">
            <Title> Input description for task B </Title>
        </Input>
        <Input port="mInDescrC" type="persistent">
            <Title> Input description for task C </Title>
        </Input>
        <Input port="mInitCost" type="temporary">
            <Title> Initial cost (should be large) </Title>
        </Input>
    </InputList>
    <OutputList>
        <Output port="mOutReport" type="persistent">
            <Title> Report of parabola evaluation </Title>
        </Output>
    </OutputList>
    <TaskList>
        <Begin/>
        <Task instance="(InitA)"
                      taskRef="parabola_defn.cdt#pInitSolver"/>
        <Task instance="(InitB)"
                      taskRef="parabola_defn.cdt#pInitSolver"/>
        <Task instance="(InitC)"
                      taskRef="parabola_defn.cdt#pInitSolver"/>
        <Task instance="(A)"
                      taskRef="parabola_defn.cdt#pSolverFlow"/>
        <Task instance="(B)"
                      taskRef="parabola_defn.cdt#pSolverFlow"/>
        <Task instance="(C)"
                      taskRef="parabola_defn.cdt#pSolverFlow"/>
        <Task instance="(D)"
                      taskRef="parabola_defn.cdt#pEvaluator"/>
        <Task instance="(E)"
                      taskRef="parabola_defn.cdt#pReport"/>
        <End/>
    </TaskList>
    <TaskGraph>
        (BEGIN) => (InitA) => (A) => (D) => (E) => (END)
                       (A) => (A)    (D) +> (A)
        (BEGIN) => (InitB) => (B) => (D) => (E) => (END)
                       (B) => (B)    (D) +> (B)
        (BEGIN) => (InitC) => (C) => (D) => (E) => (END)
                       (C) => (C)    (D) +> (C)
    </TaskGraph>
</MultiTaskDefn>
        ...
        ...
<SingleTaskDefn name="pReport"
             bodyRef="parabola_body.cdt#pReport">
    <Title> Parabola Report Generator </Title>
    <Description>
        ...
    </Description>
    <InputList>
        <Input port="totalCost" type="persistent">
            <Title> Total cost </Title>
        ...
        ...
        </Output>
    </OutputList>
</SingleTaskDefn>
</Cdtml>
<!-- parabola_defn.cdt -->
```

```
**************   parabola_main.cdt   ***************

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CDTML SYSTEM
          "http://www.cbl.ncsu.edu/OpenCdt/cdtml_schema.xml">
<!-- parabola_main.cdt -->
<Cdtml>

<MainTask>
    <TaskList>
        <Begin/>
        <Task instance="(parabola)"
               taskRef="parabola_defn.cdt#parabola"/>
        <End/>
    </TaskList>

    <TaskGraph>
        (BEGIN)  =>  (parabola)  =>  (END)
    </TaskGraph>

    <TaskInstance instance="(parabola)">
        <SetInput port="emailaddr">
            <LocalValue> lavana@cbl.ncsu.edu </LocalValue>
        </SetInput>
        <SetInput port="mInDescrA">
            <LocalValue> pbInDescrA.d </LocalValue>
        </SetInput>
        <SetInput port="mInDescrB">
            <LocalValue> pbInDescrB.d </LocalValue>
        </SetInput>
        <SetInput port="mInDescrC">
            <LocalValue> pbInDescrC.d </LocalValue>
        </SetInput>
        <SetInput port="mInitCost">
            <LocalValue> 1e6 </LocalValue>
        </SetInput>
        <SetOutput port="mOutReport">
            <LocalValue> parabola.rpt </LocalValue>
        </SetOutput>
    </TaskInstance>
</MainTask>

</Cdtml>
<!-- parabola_main.cdt -->
```

This example illustrates the recommended file organization of a taskflow description named 'parabola'. There are a total of three files:

- parabola_main.cdt (shown in its entirety above),
- parabola_defn.cdt (two representative segments are shown on the left, more in the Appendix, Figure 7), and
- parabola_body.cdt (several representative segments are shown in the Appendix, Figure 8).

Here, parabola_main.cdt invokes a task instance *(parabola)* referenced as "parabola_defn.cdt#parabola". Data entries such as variable value 'lavana@cbl.ncsu.edu', input file name 'pbInDescrA.d',..., output file name 'parabola.rpt' are passed on to the corresponding meta ports declared in "parabola_defn.cdt#parabola".

Note that in parabola_defn.cdt, parabola represents a multi-task definition layer, supported as an element of *complex type* with name *MultiTaskDefn* in the cdtML schema in Figure 3. The correspondence between the elements of the cdtML schema and the user-entered description of the multi-task definition is clear. A simple syntax describes the task graph, the symbol `=>` represents the closed state of an *InvocationEdge* or *RepeatInvocationEdge*, the symbol `+>` represents the closed state of an *AbortInvocationEdge*.

Each task definition also references its body description: e.g.

```
<MultiTaskDefn name="parabola"
            bodyRef="parabola_body.cdt#parabola">
```

Figure 5: Example of taskflow 'definition' and 'main' files written in cdtML.

most elaborate. Only a representative segment of the multi-task body layer for "parabola" is shown here, whereas a complete description of the corresponding definition layer is shown Figure 5. However, Figure 8 does contain complete descriptions of single-task body layers for the encapsulated blackbox components `pSolverWeb`, `download`, and `pEvaluator`. We highlight some of the features of these descriptions next.

MultiTaskBody name="parabola". The structure of this description follows the *MultiTaskBody* schema shown in Figure 1: full description of task instances delimited by `<BeginFork/>` and `<EndJoin/>`, followed by a data graph description delimited by `<DataGraph>` and `</DataGraph>`.

TaskInstance instance="(A)". The structure of this description follows the *TaskInstance* schema shown in Figure 1: *local* data input and data output ports for this task instance are delimited by `<DataMux>` and `<DataMux/>`. Neither ControlJoin (`<JoinConditions>` nor ControlFork (`<ForkConditions>` are entered, hence default conditions apply. Note however the explicit entry of the `<RepeatCondition>`. The reference to the encapsulated task is implicit in the instance name "(A)"; looking back to the definition file in Figure 5, we find

```
<Task instance="(A)"
       taskRef="parabola\_defn.cdt#pSolverFlow"/>
```

which points to

```
<Task instance="(pSolverWeb)"
       taskRef="parabola_defn.cdt#pSolverWeb"/>
```

which points to

```
<SingleTaskDefn name="pSolverWeb"
             bodyRef="parabola_body.cdt#pSolverWeb">
```

which is described later in this column. Another feature of interest in the description of this task instance is the use and the syntax of the taskflow-defined commands *evalRepeatCondition* and *cdtReadData*:

```
<RepeatCondition>
   <UserRepeat>
        evalRepeatCondition \
           "[cdtReadData newCost]"
                IsLessThan "[cdtReadData oldCost]"
   </UserRepeat>
</RepeatCondition>
```

TaskInstance instance="(D)". The interesting feature of this task instance description is the use and the syntax of the taskflow-defined procedure *evalJoinCondition*:

```
<JoinCondition>
  <UserInvoke>
    evalJoinCondition OR \
      .....
     [evalJoinCondition AND {(A) Valid} {(B) Valid}
                            {(C) Skipped}]        \
     [evalJoinCondition AND {(A) Valid} {(B) Skipped}
                            {(C) Valid}]          \
      .....
  </UserInvoke>
</JoinCondition>
```

The syntax shown above, while completely general, may not be suitable to express conveniently all conditions under which to invoke a task. An example of a synchronizing condition where a shorter syntax is preferred is shown below:

```
<JoinCondition>
  <UserInvoke>
     evalJoinCondition MINIMUM 2 \
             {(A) Valid} {(B) Valid} {(C) Valid}
  </UserInvoke>
</JoinCondition>
```

<DataGraph>. The DataGraph associated with the *MultiTaskBody name="parabola"* in in Figure 8 has a simple syntax, similar to the one described for the TaskGraph in Figure 4. It consist of *data port pairs*, connected by a directed DataEdge represented with a symbol `->`, e.g.

```
<DataGraph>
    emailaddr      ->  (InitA).emailaddr
    emailaddr      ->  (InitB).emailaddr
    emailaddr      ->  (InitC).emailaddr
    .........
    (D).costAvg    ->  (E).costAvg
    (D).count      ->  (E).count
    (E).outReport  ->  mOutReport
</DataGraph>
```

To appreciate the simplicity of this syntax, consider the description of parabola_main.cdt in Figure 5. Here, the value assigned to *emailaddr* is *lavana@cbl.ncsu.edu*. According to DataGraph, this value is passed to data ports of three task instances: *(InitA).emailaddr*, *(InitB).emailaddr*, *(InitC).emailaddr*. Similarly, data values computed by task instance (D) are passed to data ports of task instance (E), i.e. to *(E).costAvg* and *(E).costAvg*. The report file generated by task instance E is passed on to the data port of parabola_main and is accessible by the name of the variable, assigned to *mOutReport*: parabola.rpt in this case.

SingleTaskBody name="pSolverWeb". This blackbox component uses the exec command of type "HttpCmd" where the program `cdtHttpSubmit` is invoked with a number of parameters, ranging from URL (parabolaurl) to problem-specific variable values and files. Note also the use of the command `cdtGetData` to access values of respective data ports.

SingleTaskBody name="download". This blackbox component also uses the exec command of type "HttpCmd" where the program cdtHttpGet is invoked with a number of parameters, including URL, to download data from a web-based directory to a designated directory on the local host.

SingleTaskBody name="pEvaluator". This blackbox component uses the exec command of type "TclCmd" where the program `pEvaluator.tcl` is invoked with a number of parameters.

3. TASKFLOW GUI AND INTERACTIONS

A generic schema and the specific architecture that implements the taskflow GUI, scheduling, and execution are outlined in Section 1 and illustrated in Figure 1. The XML-based implementation of the cdtML taskflow schema and detailed examples of taskflow descriptions in cdtML are presented in Section 2. Notably, we argue that the entire list of 10 taskflow GUI features, formulated as requirements to support a taskflow-oriented programming paradigm in [2], is covered by the implementations as presented in this and the preceding sections of this paper.

In this section, we expand on the implementation of the taskflow GUI environment and the number of ways user can interact with such an environment, first shown in Figure 2.

3.1 Taskflow GUI

When organizing computational tasks in a number of projects with distributed software components, two generic views about such organizations emerged already in the conceptual phase [1, 2]. 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 at each level of hierarchy. This organization is implemented by the taskflow architecture in Figure 1; we now expand on the 'parabola' taskflow example introduced in Figure 2 and described, in cdtML, in Section 2.

Our current implementation of the taskflow GUI has four main views: a *selector view*, *tree view*, a *graph view*, and a *status view*. These views are briefly described in Figure 6; a more detailed description is given below.

Selector View. There are three modes in which user can execute the taskflow: *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, in the field labeled as 'Sleep', an integer number to indicate the number of seconds a task is to stay in the sleep state. 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.

The *View/Edit* buttons provide selections to either view or edit data nodes associated with each task. Data may be viewed or edited when clicking on data nodes that are displayed with each task instance in the graph view.

The selector view in Figure 6 displays the choice of a 'simulation mode' with an integer assignment of 2 seconds for each task to stay in the 'sleep mode' during the taskflow simulation.

Tree view. Initially, the tree view consists solely of the main task instance displayed as the root of the tree view. The children of the main task instance can be opened or closed by the user by clicking on a '+' or a '-' symbol located near the task instance node.

On opening the main task instance, it displays the data I/O port lists, 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. Only after clicking on the displayed task instance is the cdtML data parsed and translated to TclTk which renders the expanded display of the tree.

Each task instance is a button widget that user clicks to invoke and execute. There are additional control buttons in the row defined by the task instance button: *Skip*, *Exec*, *Abort*, and *Clean*. If the task instance represents a whitebox, i.e. it contains other task instances, there is also a *Load-Graph-View* button. Clicking this button renders the graph view of the task instance. We provide more details about these button when presenting the graph view.

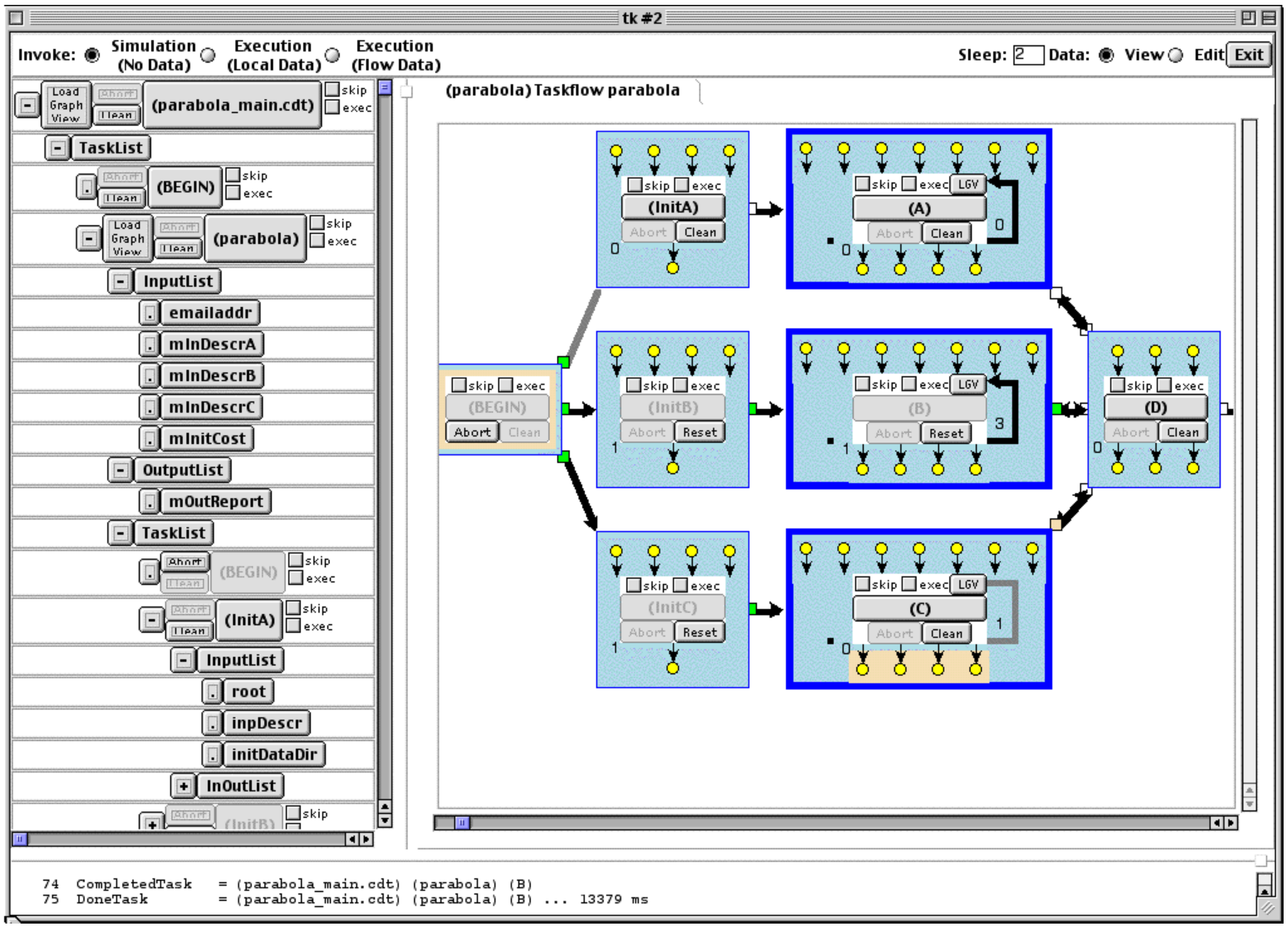
In general, the tree view provides a simple, compact user-interface for browsing the hierarchy of the task instances as well as for its interactive execution.

The tree view in Figure 6 displays a partial expansion of the taskflow tree representation. Tasks instances that are displayed as shaded, e.g. *(BEGIN)*, indicate the completion state of the task after taskflow execution; i.e. they cannot be re-invoked until user resets them.

Graph View. The graph view presents a dynamically generated and a highly interactive interface that depicts task-to-task, data-to-task and task-to-data dependencies at a given level of taskflow hierarchy. Elements of each task instance are encapsulated in a shaded box as shown; a thin edge represents an instance of a blackbox component, a thick edge represents an instance of a whitebox component. Each task instance contains the following elements with which user can interact:

- a *task button* that user can click to invoke the task instance directly (rather than wait for the task to be invoked and executed by other tasks in accordance with a schedule).
- 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.
- 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. This button becomes a *Reset* button after a valid completion of the task.
- a *LGV* button for descending the taskflow hierarchy, i.e. loading another graph view if the task instance represents a whitebox.
- *data nodes*, represented as circles and connected with *data-edges* to the input, inout, outin and output ports of the respective task instance. Data may be viewed or edited when clicking on any data node.
- an optional *RepeatInvocationEdge* whose state can be toggled between open/closed with a user click. This edge is displayed only if user specified it in the cdtML description of the taskgraph.

A taskgraph is formed by connecting the task instances with *InvocationEdges* or *AbortInvocationEdges* as described in the earlier section. User can click on any of these edges to select or de-select task instances for invocation by other task instances, thereby controlling the invocation schedule, generated dynamically at the runtime.



```
  74  CompletedTask       = (parabola_main.cdt) (parabola) (B)
  75  DoneTask            = (parabola_main.cdt) (parabola) (B) ... 13379 ms
```

There are four main views of the taskflow GUI:

Selector View: The *Invocation Mode* buttons provide three choices: *simulation, execution with local data, execution with flow data.* The *View/Edit* buttons provide selections to view/edit data nodes of each task. In the simulation mode, the *Sleep* entry assigns a random/fixed sleep interval for each task.

Tree View: The tree view provides an intuitive, dynamically generated interface to browse the hierarchy of task instances, including data I/O definitions. Each task instance is a button that user may click to invoke and execute. Clicking the *Load-Graph-View* button next to each instance of a *whitebox* renders its graph view.

Graph View: The graph view is a dynamically generated and a highly interactive interface that depicts task-to-task, data-to-task and task-to-data dependencies. User clicks on *Invocation, RepeatInvocation,* or *AbortInvocation* edges to select or de-select task instances which in turn control the execution schedule, generated dynamically at the runtime. User click on the task button also invokes the task instance. Using a color scheme, current state of each task is displayed in the tail box of the Invocation/AbortInvocation edge. See text for more descriptions of these views.

Status View: Messages generated during user interactions *and* taskflow executions are captured in the status view. The simulation log on the right illustrates that the selected task instances are indeed *executing concurrently.*

```
 ==  parabola-web  ==
UserInvokeTask:     (main..) (p..) (BEGIN)
 1  BeginForkTask = (main..) (p..) (BEGIN)
 2  EnabledTask   = (main..) (p..) (InitB)
 3  EnabledTask   = (main..) (p..) (InitC)
 4  ExecutingTask = (main..) (p..) (InitB)
 5  Sleep 2000 ms = (main..) (p..) (InitB)
 6  ExecutingTask = (main..) (p..) (InitC)
 7  Sleep 2000 ms = (main..) (p..) (InitC)
 8  CompletedTask = (main..) (p..) (InitB)
 9  CompletedTask = (main..) (p..) (InitC)
10  DoneTask      = (main..) (p..) (InitB) .. 2078 ms
11  DoneTask      = (main..) (p..) (InitC) .. 2089 ms
12  EnabledTask   = (main..) (p..) (B)
13  ExecutingTask = (main..) (p..) (B)
14  EnabledTask   = (main..) (p..) (C)
15  ExecutingTask = (main..) (p..) (C)
16  BeginForkTask = (main..) (p..) (B) (BEGIN)
17  BeginForkTask = (main..) (p..) (C) (BEGIN)
18  EnabledTask   = (main..) (p..) (B) (pSolverWeb)
19  EnabledTask   = (main..) (p..) (C) (pSolverWeb)
20  ExecutingTask = (main..) (p..) (B) (pSolverWeb)
21  Sleep 2000 ms = (main..) (p..) (B) (pSolverWeb)
22  ExecutingTask = (main..) (p..) (C) (pSolverWeb)
23  Sleep 2000 ms = (main..) (p..) (C) (pSolverWeb)
24  CompletedTask = (main..) (p..) (B) (pSolverWeb)
25  DoneTask      = (main..) (p..) (B) (pSolverWeb) .. 2020 ms
26  CompletedTask = (main..) (p..) (C) (pSolverWeb)
27  DoneTask      = (main..) (p..) (C) (pSolverWeb) .. 2035 ms
28  EnabledTask   = (main..) (p..) (B) (download)
29  EnabledTask   = (main..) (p..) (C) (download)
.................
```

Figure 6: Key elements of the taskflow GUI design: a tree view and a graph view.

Using a color scheme, current state of each task instance is displayed in the square between the task instance boundary and the head of the Invocation/AbortInvocation edge:

- *white*, to indicate the task *waiting* state;
- *wheat*, to indicate the task *enabled/executing* state;
- *green*, to indicate the task ForkCondition *valid* state;
- *violet*, to indicate the task ForkCondition *not-valid* state;
- *yellow*, to indicate the task *skipped* state.
- *pink*, to indicate the task *timed-out* state;
- *red*, to indicate the task *aborted* state.

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

The graph view in Figure 6 displays task instances that have not been invoked and are in waiting states (tasks *(InitA), (A), (D)*), tasks that have been invoked and completed execution as *valid* (tasks *(BEGIN), (InitB), (InitC), (B)*), and tasks that have been invoked and are still in *executing* state (task *(C)*). In the current implementation, the AbortInvocationEdges from task (D) overlap with the incoming InvocationEdges from tasks (A), (B), and (C).

Status View. Messages generated during user interactions *and* taskflow executions are captured in the status view. User can scroll through the entire set of such messages during the session. This view can also assist in debugging and restructuring a taskflow composition.

The status view in Figure 6 displays the last few states of the taskflow before its execution came to a halt. A textbox below this view display the first 29 states of the taskflow when executing in the simulation mode. The trace clearly illustrates that the selected task instances are indeed *executing concurrently.*

3.2 Taskflow Interactive Environment

The basic premise of taskflow-oriented programming with OmniFlow is that user directly interacts with an environment in which he controls the linking, scheduling, and execution of its components. The underlying cdtML configuration is composed with a text editor or more conveniently, a validating XML editor. The environment itself is created by OmniFlow that parses and renders the cdtML configurations as the interactive taskflow environment. As anticipated in [2], the list of point-and-click features includes:

- open, close, ascent, and descent of the tree and the graph hierarchy.
- reconfiguration of a invocation, repeat invocation, and abort invocation edge into a 'closed' or 'open' state.
- invocation of the taskflow and its schedule by clicking on any of its task nodes, including the *Begin* task node.
- abort of the taskflow by clicking on one or more of the executing task nodes, propagated in a descending order of taskflow hierarchy.
- reset of the taskflow state, propagated in a descending order of taskflow hierarchy.
- access to view and edit data associated with each task, represented as input (output) data nodes associated with each task.

To illustrate but a small fraction of such interactivity, consider the steps to reach the state of the taskflow in Figure 6 *after* its initial rendering:

- *in the selector view*, choose the simulation mode and enter the value of 2;
- *in the tree view*, expand the task instance *parabola_main*, find task instance *parabola* and click on the *Load-Graph-View* button in the same row;
- *in the tree view*, change the states of two edges from 'closed' to 'open': the edge from *(BEGIN)* to *(InitA)*, and the edge that forms a self-loop with task *(C)*;
- *in the tree view*, click on task instance *(BEGIN)* and observe the flow of execution: tasks *(InitA)* and *(InitB)* invoke concurrently, and after about 2 seconds, tasks *(InitB)* and *(InitC)* invoke concurrently, each invoking (B) and (C) respectively, i.e. the different instances of tasks *(pSolverWeb)* and *(download)*, each taking 2 seconds to 'sleep' for a nominal total of 4 seconds for (B), and 4 seconds for (C).
- *in the status view*, the 'halted' taskflow state (#75) reports a total of 13.379 seconds execution time for instance (B). Nominally, this task repeated 3 times and would have reported $3 \times (2 + 2) = 12$ seconds – if there were no overhead in scheduling and executing the tasks.

The section that follows, reports a number of performance experiments that reveal a consistent and efficient performance of the taskflow scheduler.

4. PERFORMANCE EXPERIMENTS

Comprehensive experiments with a variety of taskflows have already demonstrated the efficiency of the GUI implementation and the scheduler [1]. Rendering a taskflow such as shown in Figure 6 is less than 2 seconds (under Solaris/Linux/ WindowsNT/MacOS) and user interactions such as changing the state of any invocation edges from open to closed (and vice versa) is instantaneous. 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 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.

An entire chapter in [1] is devoted to descriptions of taskflow implementations of collaborative and distributed computing projects, bringing together a number of university-based as well as commercial stand-alone programs. Experiments also include records of traces for a number of distinctive scheduling patterns: from simple sequencing of tasks, to splits, concurrency, joins, iterations, and cycles. A description of these patterns is summarized in [2].

5. SUMMARY AND CONCLUSIONS

This paper presents an XML/TclTk implementation of a universal, user-configurable and highly interactive client (OmniFlow) that creates a taskflow-oriented programming environment, conceptually introduced in the companion paper [2]. In the taskflow-oriented programming paradigm, task instances represent distributed stand-alone component programs that user composes into interactive, executable programs. User alone can now (1) write a *hierarchical* taskflow configuration, (2) invoke the *universal client (OmniFlow)* that reads the configuration and renders its GUI, and (3) interact with the taskflow in a number of ways.

The recursive schema of component instances is conveniently captured as an extension of XML in a collaborative distributed task mark-up language cdtML. 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.

Experimental evaluations of the client prototype on a number of networked design and computing projects, including taskflow descriptions with up to 9150 tasks executing serially *and* concurrently on the longest path of 1600 tasks, demonstrate the scalability of the environment and the overall effectiveness of the proposed architecture. A user guide and a cross-platform software prototype of the client described in this paper will be posted by June 2001 under

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

Acknowledgment. The implementation of OmniFlow benefited from the on-going cross-platform support for TclTk by the Tcl core team and other contributors [14]. In particular, we acknowledge the use of the following cross-platform libraries and packages: tcllib0.8, tclxml2.0, bwidget1.3. Also, the use of demo version of XMLSpy, a validating XML editor [16], has been most productive.

6. REFERENCES

[1] 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`.

[2] F. Brglez and H. Lavana. A Universal Client for Taskflow-Oriented Programming with Distributed Components: Concepts. 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-Brglez`.

[3] W3C Home Page for XML Schema, September 2000. For more information, see `http://www.w3.org/XML/Schema.html`.

[4] D. Hunter et al. *Begining XML.* Wrox, 2000.

[5] XML Resource Guide, 2000. For more information, see `http://www.xml.com/resourceguide`.

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

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

[8] A. Khetawat, H. Lavana, and F. Brglez. Internet-based Desktops in Tcl/Tk: Collaborative and Recordable. In *Sixth Annual Tcl/Tk Conference.* USENIX, September 1998. Also available at `http://www.cbl.ncsu.edu/-publications/#1998-TclTk-Khetawat`.

[9] F. Brglez, H. Lavana, Z. Fu, D. Ghosh, L. I. Moffitt, S. Nelson, J. M. Smith, and J. Zhou. Collaborative Client-Server Architectures in Tcl/Tk: A Class Project Experiment and Experience. In *Seventh Annual Tcl/Tk Conference.* USENIX, February 2000. Also available at `http://www.cbl.ncsu.edu/-publications/#2000-TclTk-Brglez`.

[10] Tcl/Tk Web Browser Plug-in, 2001. See `http://tcl.activestate.com/software/plugin/`.

[11] H. Lavana and F. Brglez. WebWiseTclTk: A Safe-Tcl/Tk-based Toolkit Enhanced for the World Wide Web. In *Sixth Annual Tcl/Tk Conference (Best Student Paper Award).* USENIX, September 1998. Also available at `http://www.cbl.ncsu.edu/-publications/#1998-TclTk-Lavana`.

[12] H. Lavana and F. Brglez. CollabWiseTk: A Toolkit for Rendering Stand-alone Applications Collaborative. In *Seventh Annual Tcl/Tk Conference.* USENIX, February 2000. Also available at `http://www.cbl.ncsu.edu/publications/-#2000-TclTk-Lavana`.

[13] S. Ball. TclXML Parser, 2001. See `http://www.zveno.com/zm.cgi/in-tclxml/`.

[14] Home Page of Tcl Developer Xchange, 2001. See `http://tcl.activestate.com`.

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

[16] XML Spy, A Validating XML Editor, 2001. See `http://www.xmlspy.com/`.

[17] E.R. Gansner, E. Koutsifios, S.C. North and K.P. Vo. A Technique for Drawing Directed Graphs. *IEEE Trans. Software Engg.*, 19:214–230, 1993. See also `http://www.research.att.com/sw/tools/graphviz/`.

[18] F. Brglez and H. Lavana. OmniFlow User Guide on Taskflow-Oriented Programming and Computing with Distributed Networked Components, July 2001. Available from `http://www.cbl.ncsu.edu/-publications/#2001-UG@CBL-OmniFlow`.

APPENDIX

Figures 7 and 8 in this Appendix extend the cdtML descriptions of the taskflow 'parabola', described in Section 2.

************** parabola_defn.cdt ***************

```
.............
<MultiTaskDefn name="pSolverFlow"
            bodyRef="parabola_body.cdt#pSolverFlow">
    <Title> pSolver Flow </Title>
    <Description>
        ...
    </Description>
    <InputList>
        <Input port="emailaddr" type="temporary">
            <Title> Email address </Title>
        </Input>
        <Input port="root" type="temporary">
            <Title> Root name for the task instance </Title>
        </Input>
        <Input port="inpDescr" type="persistent">
            <Title> Description of  input points </Title>
        </Input>
        <Input port="initCost" type="temporary">
            <Title> Initial Cost </Title>
        </Input>
        <Input port="initSoln" type="temporary">
            <Title> Initial solution </Title>
        </Input>
    </InputList>
    <OutInList>
        <OutIn port="newSoln" type="persistent">
            <Title> New Solution </Title>
        </OutIn>
        <OutIn port="newCost" type="persistent">
            <Title> New Cost </Title>
        </OutIn>
    </OutInList>
    <OutputList>
        <Output port="nIter" type="persistent">
            <Title> Number of Iterations </Title>
        </Output>
        <Output port="oldCost" type="persistent">
            <Title> Old Cost </Title>
        </Output>
    </OutputList>
    <TaskList>
        <Begin/>
        <Task instance="(pSolverWeb)"
               taskRef="parabola_defn.cdt#pSolverWeb"/>
        <Task instance="(download)"
               taskRef="parabola_defn.cdt#download"/>
        <End/>
    </TaskList>
    <TaskGraph>
        (BEGIN)  =>  (pSolverWeb)  =>  (download)  =>  (END)
    </TaskGraph>
</MultiTaskDefn>

<SingleTaskDefn name="pSolverWeb"
             bodyRef="parabola_body.cdt#pSolverWeb">
    <Title> Parabola Solver </Title>
    <Description>
        ...
    </Description>
    <InputList>
        <Input port="emailaddr" type="temporary">
            <Title> Email address </Title>
        </Input>
        <Input port="parabolaurl" type="temporary">
            <Title> Url for the parabola cgi-script </Title>
            <DefaultValue>
                http://www.cbl.ncsu.edu/vela/
                       coPI-only/cgi-bin/pub/Parabola
            </DefaultValue>
        </Input>
        <Input port="root" type="temporary">
            <Title> Root name for the task instance </Title>
        </Input>
        <Input port="inpDescr" type="persistent">
            <Title> Description of  input points </Title>
        </Input>
```

```
        <Input port="initCost" type="temporary">
             .....................
        </Input>
    </InputList>
    <OutInList>
        <OutIn port="newSoln" type="persistent">
            <Title> New Solution </Title>
        </OutIn>
        <OutIn port="newCost" type="persistent">
            <Title> New Cost </Title>
        </OutIn>
    </OutInList>
    <OutputList>
        <Output port="results" type="persistent">
            <Title> Save results of the http submit </Title>
        </Output>
        <Output port="nIter" type="persistent">
            <Title> Number of Iterations </Title>
        </Output>
        <Output port="oldCost" type="persistent">
            <Title> Old Cost </Title>
        </Output>
    </OutputList>
</SingleTaskDefn>

<SingleTaskDefn name="download"
             bodyRef="parabola_body.cdt#download">
    <Title> Download Files </Title>
    <Description>
        ...
    </Description>
    <InputList>
        <Input port="emailaddr" type="temporary">
            <Title> Email address </Title>
        </Input>
        <Input port="resultsurl" type="temporary">
            <Title> Url for the saved location of
                                  the results </Title>
            <DefaultValue>
             http://www.cbl.ncsu.edu/vela/SavedData/
                           [cdtGetData emailaddr]/Parabola
            </DefaultValue>
        </Input>
        <Input port="newSolnWeb" type="persistent">
            <Title> New Solution </Title>
        </Input>
        <Input port="newCostWeb" type="persistent">
            <Title> New Cost </Title>
        </Input>
        <Input port="nIterWeb" type="persistent">
            <Title> Number of Iterations </Title>
        </Input>
        <Input port="oldCostWeb" type="persistent">
            <Title> Old Cost </Title>
        </Input>
    </InputList>
    <OutputList>
        <Output port="newSolnLcl" type="persistent">
            <Title> New Solution </Title>
        </Output>
        <Output port="newCostLcl" type="persistent">
            <Title> New Cost </Title>
        </Output>
        <Output port="nIterLcl" type="persistent">
            <Title> Number of Iterations </Title>
        </Output>
        <Output port="oldCostLcl" type="persistent">
            <Title> Old Cost </Title>
        </Output>
    </OutputList>
</SingleTaskDefn>

</Cdtml>
<!-- parabola_defn.cdt -->
```

Figure 7: Example of a taskflow ‘definition’ file written in cdtML.

```
 **************   parabola_body.cdt   ***************

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE CDTML SYSTEM
         "http://www.cbl.ncsu.edu/OpenCdt/cdtml_schema.xml">
<!-- parabola_body.cdt -->
<Cdtml>

<MultiTaskBody name="parabola" sleep="">
    <BeginFork/>
    <TaskInstance instance="(InitA)"  host="h1">
        <DataMux>
            <SetInput port="root">
                <LocalValue> taskA </LocalValue>
            </SetInput>
        </DataMux>
    </TaskInstance>
    ...............
    <TaskInstance instance="(A)"  maxIterate="3" host="h1">
       <JoinCondition>
           <UserAbort>
               evalAbortCondition OR {(D) Enabled}
                                      {(D) Executing}
           </UserAbort>
       </JoinCondition>
        <DataMux>
            <SetInput port="root">
                <LocalValue> taskA </LocalValue>
            </SetInput>
            <SetInput port="inpDescr">
                <LocalValue> taskA_inpDescr.d </LocalValue>
            </SetInput>
            <SetInput port="initCost">
                <LocalValue> 1.e6 </LocalValue>
            </SetInput>
            <SetInput port="initSoln">
                <LocalValue> 5 </LocalValue>
            </SetInput>
            <SetOutIn port="newCost">
                <LocalValue> taskA_newCost.d </LocalValue>
            </SetOutIn>
            <SetOutIn port="newSoln">
                <LocalValue> taskA_newSoln.d </LocalValue>
            </SetOutIn>
            <SetOutput port="nIter">
                <LocalValue> taskA_nIter.d </LocalValue>
            </SetOutput>
            <SetOutput port="oldCost">
                ......................
        </DataMux>
        <RepeatCondition>
            <UserRepeat>
                evalRepeatCondition \
                    "[cdtReadData newCost]"
                             IsLessThan "[cdtReadData oldCost]"
            </UserRepeat>
        </RepeatCondition>
    </TaskInstance>
    ...............
    <TaskInstance instance="(D)">
        <JoinCondition>
          <UserInvoke>
            evalJoinCondition OR \
              [evalJoinCondition AND {(A) Valid} {(B) Valid}
                                     {(C) Valid}]           \
              [evalJoinCondition AND {(A) Valid} {(B) Valid}
                                     {(C) Skipped}]         \
              [evalJoinCondition AND {(A) Valid} {(B) Skipped}
               ...................................          \
              [evalJoinCondition AND {(A) Skipped} {(B) Valid}
                                     {(C) Skipped}]         \
              [evalJoinCondition AND {(A) Skipped}
                                  {(B) Skipped} {(C) Valid}] ;
          </UserInvoke>
        </JoinCondition>
        <DataMux>
        ..........
        </DataMux>
    </TaskInstance>
    <EndJoin/>
```

```

    <DataGraph>
        emailaddr      ->  (InitA).emailaddr
        emailaddr      ->  (InitB).emailaddr
        emailaddr      ->  (InitC).emailaddr
        mInDescrA      ->  (InitA).inpDescr
        mInDescrB      ->  (InitB).inpDescr
        mInDescrC      ->  (InitC).inpDescr
        (InitA).emailaddr      ->  (A).emailaddr
        (InitB).emailaddr      ->  (B).emailaddr
        (InitC).emailaddr      ->  (C).emailaddr
        mInDescrA      ->  (A).inpDescr
        mInDescrB      ->  (B).inpDescr
        mInDescrC      ->  (C).inpDescr
        mInitCost      ->  (A).initCost
        mInitCost      ->  (B).initCost
        mInitCost      ->  (C).initCost
        (A).newCost    ->  (D).costA
        (B).newCost    ->  (D).costB
        (C).newCost    ->  (D).costC
        (D).totalCost  ->  (E).totalCost
        (D).costAvg    ->  (E).costAvg
        (D).count      ->  (E).count
        (E).outReport  ->  mOutReport
    </DataGraph>
</MultiTaskBody>
................
................


<SingleTaskBody name="pSolverWeb" sleep="">
    <ExecCommand type="HttpCmd">
        <Value>
            cdtHttpSubmit  "[cdtGetData parabolaurl]"  \
                 "EmailAddr     [cdtGetData emailaddr] \
                  inpDescr      [cdtGetData inpDescr]  \
                  initCost      [cdtGetData initCost]  \
                  initSoln      [cdtGetData initSoln]  \
                  newCost       [cdtGetData newCost]   \
                  newSoln       [cdtGetData newSoln]   \
                  oldCost       [cdtGetData oldCost]   \
                  nIter         [cdtGetData nIter]"    \
                 "[cdtGetData results]"
        </Value>
    </ExecCommand>
</SingleTaskBody>


<SingleTaskBody name="download" sleep="">
    <ExecCommand type="HttpCmd">
        <Value>
         cdtHttpGet
           [cdtGetData resultsurl]/[cdtGetData oldCostWeb tail]
         cdtHttpGet
           [cdtGetData resultsurl]/[cdtGetData newCostWeb tail]
         cdtHttpGet
           [cdtGetData resultsurl]/[cdtGetData newSolnWeb tail]
         cdtHttpGet
           [cdtGetData resultsurl]/[cdtGetData nIterWeb tail]
        </Value>
    </ExecCommand>
</SingleTaskBody>


<SingleTaskBody name="pEvaluator" sleep="" host="h4">
    <ExecCommand type="TclCmd">
        <Value>
            pEvaluator.tcl
              "[cdtGetData costA]"   "[cdtGetData costB]"     \
              "[cdtGetData costC]"   "[cdtGetData totalCost]" \
               "[cdtGetData costAvg]" "[cdtGetData count]"
        </Value>
    </ExecCommand>
</SingleTaskBody>


</Cdtml>
<!-- parabola_body.cdt -->
```

Figure 8: Example of a taskflow 'body' file written in cdtML.