iTasks 2: iTasks for End-users

Bas Lijnse[1,2] and Rinus Plasmeijer[1]

[1] Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, the Netherlands
[2] Faculty of Military Sciences,
Netherlands Defense Academy, Den Helder, the Netherlands
{b.lijnse,rinus}@cs.ru.nl

Abstract. Workflow management systems (WFMSs) are systems that generate, coordinate and monitor tasks performed by human workers in collaboration with automated (information) systems. The iTask system (iTasks) is a WFMS that uses a combinator language embedded in the pure and lazy functional language Clean for the specification of highly *dynamic* workflows. iTask workflow specifications are *declarative* in the sense that they only specify (business) processes and the types of data involved. They abstract from user interface and storage issues, which are handled *generically* by the workflow engine.
Earlier work has focused on the development of the iTask combinator language. The workflow language was implemented as an engine that evaluated task combinator expressions and generated interactive web pages. Although suitable for its original purpose, this architecture has proven to be less so for generating practically usable workflow support systems. In this paper we present a new implementation of the iTask system that implements the combinator library using a service based architecture that exposes the workflow and a user friendly Ajax client. Because user interface issues are outside the scope of workflow specifications, and cannot be specified explicitly, it is crucial that the generic operationalization of the declarative interaction primitives is of adequate quality. We explain the novel generic libraries we have developed for this purpose.

1 Introduction

Workflow management systems (WFMSs) are systems that generate, coordinate and monitor tasks performed by human workers in collaboration with automated (information) systems. Many contemporary WFMSs suffer from lack of flexibility. This is partially caused by the *static* nature of the languages used for modeling the business processes they coordinate. To address this limitation the iTask system has been developed. This system uses a function combinator library embedded in the pure and lazy functional programming language Clean to model business processes, and allows specification of highly dynamic workflows. The iTask system uses *declarative* specifications of tasks. Task specifications define what has to be done, by whom and when. However, they do not specify how tasks are presented to users, how results are entered, or how progress is visualized. These operational details are taken care of fully automatically.

Earlier work [6, 9, 11, 12] has focused primarily on the benefits of the iTask system for programmers. Its goal has been to develop and extend the iTask combinator library to be able to express powerful, yet concise, specifications of arbitrary business processes. For this purpose a prototype implementation of the iTask engine with a minimum level of usability that could be used to simulate workflow scenarios by expert users has been sufficient.

In this paper we present a new implementation of the iTask system that uses a service based architecture to enable a practically applicable interface for end-users. Since user interaction is considered a declarative aspect of the iTask language and outside the scope of a workflow specification, it is *critical* for the usefulness of the iTask system that the generic framework performs adequately in this area. We show how we operationalize workflow specifications in such a way that, for end-users, selecting and working on tasks is no more difficult than the use of an average e-mail client.

The contributions of this paper are the following:

– We present a new implementation of the iTask system. We discuss its new service based architecture and key features, and how it compares to previous implementations.
– We explain the declarative nature of the iTask system. We discuss *what is* specified by iTask expressions, and *what is not*. We show *how* workflow specifications are operationalized by the iTask engine.
– We present a novel generic web interface library in Clean. This library provides *type-driven* Html visualizations of data as well as editable Ajax forms for manipulating data.

The remainder of this paper is organized as follows: First we cover the concept of declarative workflow specification in the iTask system in Section 2. Then an architectural overview of the iTask system is given in Section 3. The generic web-interface library is explained in Section 4. We discuss related work in Section 5 after which final concluding remarks are given in Section 6.

2 Declarative Workflow Specification

The iTask combinator language is designed for declarative specification of workflows. This means that the specifications describe *what* has to be done, not *how*. However, one cannot speak of a language being declarative without specifying at which level of granularity. The level of abstraction of a domain determines whether a specification can be classified as declarative at that level. Since this level is not always immediately clear, especially in workflow languages, we elaborate on it some more in this section.

2.1 When Is a Workflow Specification Declarative?

The iTask system is based on the idea that in workflow support systems, the only differences that really matter between two systems are: 1) The (business) process they support, and 2) The data that is exchanged between actors. Everything else that is needed to build these systems can be generic. The iTask system provides both a specification language to describe the processes and data, as well as a framework that provides the generic foundation that operationalizes them.

In this context, we classify a specification as declarative when everything in it specifies either data or process. Contrary to what is sometimes called declarative workflow, a process can be specified very rigidly but still be considered declarative with respect to this definition. A specification that also specifies issues such as presentation, or storage is considered not declarative in this context. A quick glance at the signature of one of the iTask primitives for interacting with users in Figure 1 illustrates this best. For instance, the `enterInformation` primitive yields a task that asks a user to provide some information. This primitive describes the action that is needed to achieve some goal, but leaves entirely open *how* information is entered.

2.2 The iTask Workflow Language

Above we have already loosely mentioned the iTask specification language, yet we have not explained how it is defined and implemented. The iTask language is a domain specific language embedded in the pure and lazy functional programming language Clean. It is essentially an API of functions and (monadic) function combinators that is used to construct complex functions that when evaluated compute the tasks that have to be done. However, from the point of view of a workflow programmer, the combinator API is just a collection of primitives and operators that are used to define workflows in a syntax that just happens to have a striking resemblance to Clean.

The central concept of iTask workflow specifications is that everything is a task that produces a *typed* result once it is done. Tasks are represented by the abstract Clean type `:: Task a`, where `a` is the type of the result of the task. Although everything is a task, we can still make a distinction between *basic tasks* and *combined tasks*. Basic tasks are the smallest units of work like entering some data in a form, or reading a piece of data from a database. From these basic tasks, larger more complex tasks are constructed using *task combinators*. For example the monadic bind combinator (`>>=`), where the result of the first task is passed to a function that computes the second. By combining tasks sequentially, in parallel or conditionally, tasks of unlimited complexity can be constructed. A short excerpt with common tasks and combinators from the iTask API is shown in Figure 1 [3]. The full API consists of many more basic tasks and combinators, like for instance, for interacting with users, generic storage and retrieval, access to meta-data of other workflows and users. Examples of iTask workflow specifications have been given in [9, 11].

[3] Context restrictions on overloaded types have been omitted for clarity

— Basic tasks —

```
// Ask a user to enter information.
enterInformation      :: question   → Task a
// Ask a user to enter information while subject information is shown
enterInformationAbout :: question s → Task a
// Show a message to a user
showMessage           :: message   → Task Void
// Show a message and subject information to a user
showMessageAbout      :: message s → Task Void
// Create a value in the data store
dbCreateItem          ::                Task a
// Read a value from the data store
dbReadItem            :: !(DBRef a) → Task (Maybe a)
```

— Task combinators —

```
// Lift a value to the task domain
return          :: a                      → Task a
// Bind two tasks sequentially
(>>=)  infixl 1 :: (Task a) (a → Task b) → Task b
// Assign a task to another user
(@:)   infixr 5 :: UserId    (Task a)     → Task a
// Execute two tasks in parallel
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a,b)
// Execute two tasks in parallel, finish as soon as one yields a result
(-||-) infixr 3 :: (Task a) (Task a) → Task a
// Execute all tasks in parallel
allTasks        :: ([Task a] → Task [a])
// Execute all tasks in parallel, finish as soon as one yields a result
anyTask         :: ([Task a] → Task a)
```

Fig. 1. A short excerpt from the iTask API

2.3 Implementation Consequences

As can be seen in the API in Figure 1, workflow specifications in the iTask system define nothing more than data and process. However, a complete executable workflow system is generated from just that and nothing else. A major consequence of this design is that this generic foundation that is used to generate a working system from these high level specifications must be of such quality, that there is no need to further hack or tweak the system after generation. When this is not the case the risk exists that clever programmers will find ways to abuse the workflow language to force for example a specific interface layout. This clutters the workflow definitions and makes them no longer declarative.

Of course there are domains where generic solutions are far inferior to specialized instances. Entering a location for example, is easier by putting a marker on a map than by entering coordinates in a form. For these situations the iTask system provides the possibility to define custom domain libraries that contain data types and task primitives along with specializations of the generics. This enables the use of custom code when necessary without cluttering the workflow specifications.

3 The Revised iTask System

As mentioned in Section 1 the original iTask system was used primarily to explore the design of a workflow language based on function combinators. However, experiments with building applications beyond the level of toy examples showed that much hacking and tweaking was necessary to build somewhat usable applications. Examples of such tweaking are: the use of multiple variants of essentially the same task: `chooseTaskWithButtons` and `chooseTaskWithRadios`, or the use of presentation oriented data types such as `HtmlTextArea` instead of just `String`. To be able to generate iTask applications at the level of usability that may be expected from contemporary web-based information and workflow systems, without cluttering the workflow specifications with presentation issues, a major redesign of the iTask engine was necessary.

3.1 Original Architecture

Originally the architecture of the iTask system as presented in [10, 9] was that of a simple web application that dynamically generated Html pages. The content of these pages was generated by a program compiled from an iTask workflow specification and a generic base system. This architecture is depicted graphically in the left diagram of Figure 2. Page content generation was performed by application of a workflow definition to an initial state which yielded an output state that accumulated Html code. The abstract type `Task a` of task primitives and combinators was defined as `Task a :== *TSt → (a,*TSt)` which is Clean's notation for a function that takes a unique state of type `TSt` and returns a value of type `a` and new state. Additionally to generating the Html code for the tasks to display

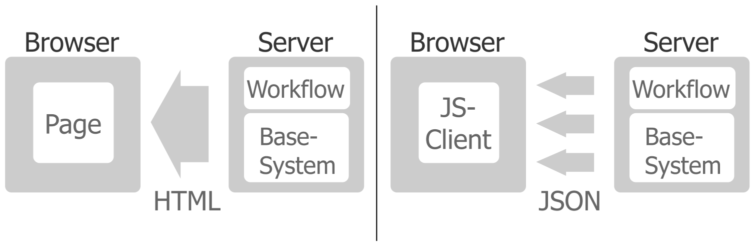


Fig. 2. Architecture old (left) and new (right) iTask system

on the page, `TSt` also accumulated ad-hoc meta-data about tasks, which was used to generate the navigation components for switching between tasks. When users triggered some event in the generated page, like clicking a button or changing the content of a textbox, the event was sent to the server by reloading the entire page, and used to generate the updated page. This was necessary because each event could potentially cause the workflow to be reduced or the user interface to be different.

3.2 Fundamental Problems

The original architecture, though suitable for showing the expressive power of the combinators, suffered from some scalability problems. When used in a more realistic setting, this architecture has a number of fundamental problems.

1. The first issue is one of separation of concerns. The original implementation of the task combinators as functions that both compute the advancement in a workflow *and* the representation of that workflow as a user interface only works for small examples. As soon as you want to define more intricate workflow combinators or put higher demands on the user interface, the implementations of the workflow combinators quickly becomes too complex to manage.
2. Another problem, which is related to the previous issue, is that in the original architecture the only way to interact with iTask workflows was through the web interface. There was no easy means of integrating with other systems. The obvious solution would be to add some flavor of remote procedure calling to the system, but this would then also have to be handled *within* the combinators, making them even more complex.
3. The final issue, which may appear trivial, is the necessity to reload an entire page after each event. This approach is not only costly in terms of network overhead, it also inherently limits the possibilities for building a decent user interface. Essential local state, such as cursor focus, is lost during a page reload which makes filling out a simple form using just the keyboard nearly impossible.

3.3 Improved Architecture

To solve the problems described in the previous section, a drastic redesign of the iTask system was needed. The only way to address them was to re-implement the iTask combinator language on top of a different architecture.

The architecture of the new iTask implementation is a web-service based client-server architecture and is shown in head to head comparion with the old architecture in Figure 2 and illustrated in more detail in Figure 3. The major difference between the old and new architecture is that the new server system does not generate web pages. Instead, it evaluates workflow specifications with stored state of workflow instances to generate datastructures called *Task Trees*. These represent the current state of workflows at the task level. These trees contain structural information: how tasks are composed of subtasks, meta-data: for example, which user is assigned to which task, and task content: a definition of work that has to be done. For interactive tasks, the content is a high-level user interface definition that can be automatically generated, which will be explained in Section 4. Task trees can be queried and manipulated by a client program through a set of JSON (JavaScript Object Notation: A lightweight data-interchange format) web services.

The overview shown in Figure 3 illustrates how the various components in the server correspond with components in the client. The workflow specifications are queried directly through the workflow directory service. The authentication service queries the user store. All other services use the task trees as intermediate representation. In the next section, the computation of task trees and the individual services are explained in more detail.

The iTask system provides a default web based Ajax client system, described in Section 3.5, that lets users browse their task list, start new workflow instances and work on multiple tasks concurrently. However, because the service based architecture nicely separates the computation of workflow state from presentation, and communication is based on open web standards, it is also easy to integrate with external systems. For example, we have also built a special purpose client written in Python that monitors a filesystem for new documents and starts a new workflow for processing that simply uses the same services as the standard client.

3.4 The Server System

The server system manages a database with the state of all active workflow instances (processes) and user and session information. It offers interaction with the workflow instances through JSON webservices. Requests to these services are HTTP requests that use HTTP POST variables to pass arguments. Responses are JSON encoded data structures. The server system is generated by compiling a Clean program that evaluates the `startEngine` function defined by the iTask base system. This function takes a list of workflow specifications as its argument. The iTask system provides two implementations of the `startEngine` function. One implements a simple HTTP server, which is useful for development and testing.

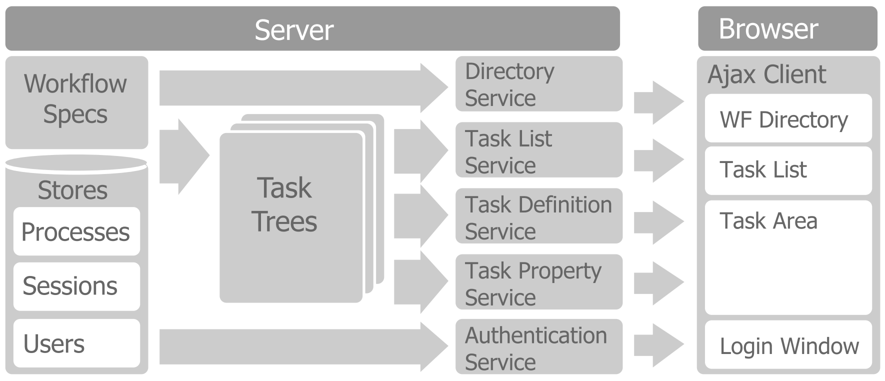


Fig. 3. A detailed architecture overview

The other implements the server system as a CGI application for use with third party web server software.

Task Tree Computation The core task of the server system is to compute and update representations of the current states of executing workflow processes. The central internal representation of the state of a workflow instance that is computed by a combinator expression is a data structure called *Task Tree*. It is a tree structure where the leaves are the atomic tasks that have to be performed, and the nodes are compositions of other tasks. It is the primary interface between the workflow specifications and the rest of the framework and is queried to generate task lists and user interface definitions. Task trees are defined by the following Clean data type:

```
:: TaskTree
  = // A stand-alone unit of work with meta-data
    TTMainTask         TaskInfo TaskProperties  [TaskTree]
    // A task composed of a sequence of tasks
  | TTSequenceTask     TaskInfo                 [TaskTree]
    // A task composed of a set tasks to be executed in parallel
  | TTParallelTask     TaskInfo                 [TaskTree]
    // A task that interacts with a user
  | TTInteractiveTask  TaskInfo (Either TUIDef  [TUIUpdate])
    // A task that monitors an external event source
  | TTMonitorTask      TaskInfo [HtmlTag]
    // A completed task
  | TTFinishedTask     TaskInfo

// Shared node information: task identifiers, labels, debug info etc.
:: TaskInfo
// Task meta-data for main tasks, assignedd user, priority etc.
```

```
:: TaskProperties
```

Every function of type `Task a` generates a (sub) task tree. Combined tasks use their argument tasks to compute the required sub task trees. Because an explanation of task tree generation is impossible without examining the combinators in detail, we will restrict ourselves to a demonstration of their use by means of an example. Let's consider the following simple workflow specification:

```
bugReport :: Task Void
bugReport = reportBug >>= fixBug
where
  reportBug :: Task BugReport
  reportBug = enterInformation "Please describe the bug you have found"

  fixBug :: BugReport → Task Void
  fixBug bug = "bas" @: (showMessageAbout "Please fix the following bug" bug)
```

Figure 4 graphically illustrates two task trees that reflect the state of this workflow at two moments during execution. The tree on the left is produced during the execution of the first `reportBug` task. The bind (`>>=`) combinator only has a left branch, which is the `TTInteractiveTask` that contains a user interface definition for the bug report form. The tree on the right is produced during the execution of `fixBug`. At this point the leftmost branch is reduced to a `TTFinishedTask` and the `@:` has been expanded to a subtree consisting of a bind of some `getUserByName` task, that is finished, and a `TTMainTask` containing the `TTInteractiveTask` with the interface definition for showing the bug report.

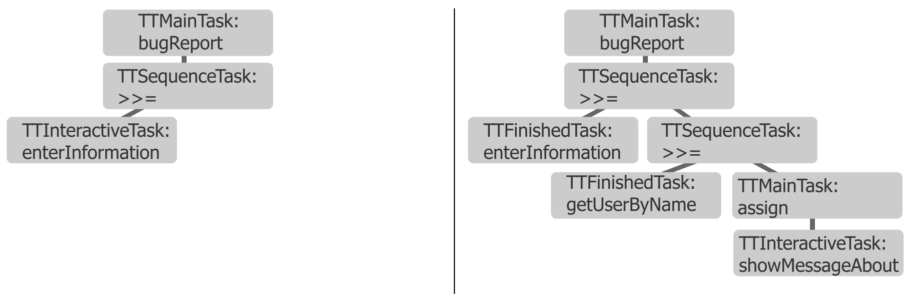

TTMainTask: bugReport
TTSequenceTask: >>=
TTInteractiveTask: enterInformation
TTMainTask: bugReport
TTSequenceTask: >>=
TTFinishedTask: enterInformation
TTSequenceTask: >>=
TTFinishedTask: getUserByName
TTMainTask: assign
TTInteractiveTask: showMessageAbout

Fig. 4. Task tree during reportBug (left) and fixBug (right)

The Authentication Service The iTask server maintains a user and role database such that (parts of) workflows can be restricted to users with special roles, and roles may be used to find the right type of worker to do a certain task. The server handles authentication of clients and keeps a database of authenticated time-limited sessions. This service consist of two methods,

`/handlers/authenticate` which accepts a username and password and yields a session key to access the other services, and `/handlers/deauthenticate` that can be passed a session key to explicitly terminate a session.

The Workflow Directory Service In order to initiate new workflow instances, the iTask server offers a directory service to browse the available workflow definitions. The server maintains a hierarchic directory of available workflows that are filtered by the roles of a user. The `/handlers/new/list` method yields the list of possible workflows and subdirectories for any given node in the hierarchy. The `/handlers/new/start` method starts a new instance of a workflow definition and returns a task identification number for the top level task of that workflow instance.

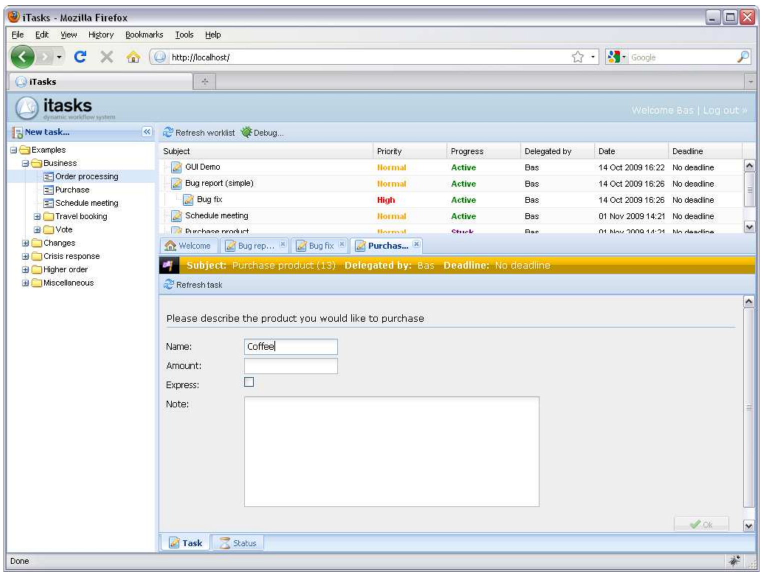
The Tasklist Service Users can find out if there is work for them through the tasklist service. The `/handlers/work/list` method yields a list of all *main tasks* assigned to the current user along with the meta-data of those tasks. This list is an aggregation of all active tasks in all workflow instances the current user is involved in. Because tasks are often subtasks of other tasks, parent/child relation information is also available in the list entries to enable grouping in a client.

The Task Service To actually get some work done, users will have to be able to work on tasks through some user interface. Because the tasks are highly dynamic, no fixed user interface can be used. Therefore, the iTask system uses a generic library to generate high-level user interface definitions that are interpreted by the client. The `/handlers/work/tab` method returns a tree structure that represents the current state of a workflow instance. This tree data is used by a client either to render an interface, or to adapt an already rendered interface. When a user updates an interactive control, this method is called with the event passed as argument. This yields a new tree that represents the updated state of the workflow after this event and possibly events from other users. This process is explained in more detail in Section 4.

The Property Service To update the meta-data of a workflow instance, for example to reassign tasks to different users or change their priority, the service `/handlers/work/property` may be used. This service can set any of the meta-data properties of a workflow instance.

3.5 The Client System

Although the iTask system focuses on workflow specification and execution on the server, the average end-user will only interact with this server through a client. While the JSON service API is not limited to one specific client, the iTask system provides a default Javascript client built with the ExtJS framework. ExtJS is a Javascript library that facilitates construction of “desktop like” Ajax



Subject	Priority	Progress	Delegated by	Date	Deadline
GUI Demo	Normal	Active	Bas	14 Oct 2009 16:22	No deadline
Bug report (simple)	Normal	Active	Bas	14 Oct 2009 16:26	No deadline
Bug fix	High	Active	Bas	14 Oct 2009 16:26	No deadline
Schedule meeting	Normal	Active	Bas	01 Nov 2009 14:21	No deadline
Purchase product	Normal	Stuck	Bas	01 Nov 2009 14:21	No deadline

Subject: Purchase product (13) **Delegated by:** Bas **Deadline:** No deadline

Please describe the product you would like to purchase

Name: Coffee
Amount:
Express: ☐
Note:

Fig. 5. The iTask client interface

applications with multiple windows, different kinds of panels, and other GUI components in a web browser. The iTask client runs in any modern webbrowser and provides everything a user needs to view and work on tasks. Figure 5 shows a screenshot of the iTask client with multiple tasks opened. The client user interface is divided into three primary areas in a layout that is common in e-mail client applications. This similarity is chosen deliberately to ease the learning of the application. The area on the left of the screen shows a folder hierarchy that accesses the workflow directory service. New workflow instances can be started by clicking the available flows in the folders. The top right area shows a user's *task list*, and the final main area is the lower right *task area*. In this part of the interface area users can work on multiple tasks concurrently in a series of tabs. New tabs are opened by clicking items in the task list.

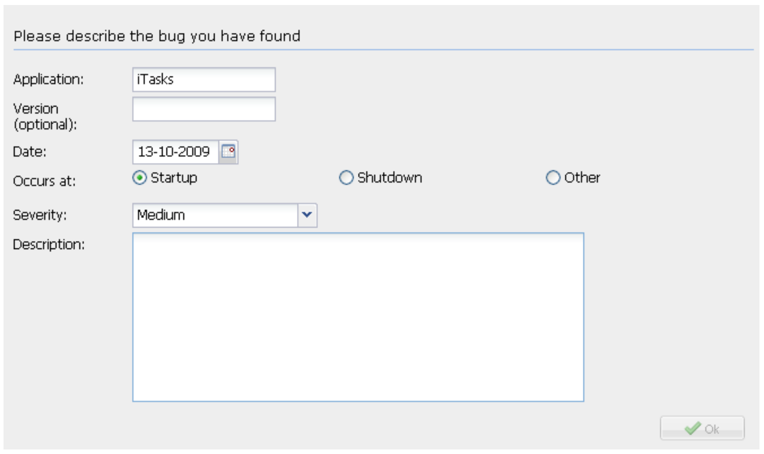
The most interesting feature of the client application is its ability to dynamically render and update arbitrary user interfaces defined by the server. It constructs the user interfaces required to work on tasks by interpreting a definition it receives from the server. It then monitors all interactive elements in the interface and synchronizes changes to them with the server in the background. The server processes the user input and responds by sending instructions to adapt the user interface when necessary. A big advantage of this design is that the server is kept synchronized with the client. This way the server can provide immediate feedback or dynamically extend an interface without the need for page refreshes. It also means that tasks can be reassigned at any moment without losing any work.

4 Dynamic Generic Web-Interfaces

One of the primary reasons for redesigning the iTask using a different architecture was to improve the user experience for end-users. In this section we show how the new iTask system makes use of the new architecture to operationalize the declaritive user interaction primitives of the specification language.

For basic tasks like `enterInformation` or `displayMessageAbout` to be operationalized, the iTask system needs to be able to generate forms for entering data and visualizations of data to display. Because user interface issues are an aspect that is abstracted from in the iTask specification language, it is *essential* that its implementation is able to generate satisfactory user interfaces. For *any* type that someone defines in a workflow specification, the system needs to be able to generate forms and renderings that have to have the following properties:

– They need to be layed out in a visually and ergonomically pleasing way.
– They need to react responsively and consistently. The cursor should follow a logical path when using the keyboard to navigate through a form and there must never be unexplainable loss or change of focus.
– They must communicate clearly what is optional and what is mandatory. The forms must ensure that mandatory input is entered.
– They must be able to adapt dynamically depending on the choice of constructor for algebraic data types. It is, for example, simply impossible to generate a static form for entering a list, because the number of elements is unbounded.


Please describe the bug you have found

Application:	iTasks
Version (optional):	
Date:	13-10-2009
Occurs at:	[✓ Startup] Shutdown Other
Severity:	Medium
Description:	

Ok

Fig. 6. An automatically generated form

The redesign of the iTask system with a service based architecture and stand-alone (Javascript) client as explained in Section 3 removes the implicit usability limitations of the original iTask system. It enables a new approach to dynamic interface generation that uses *type generic functions* as can be defined in Clean [1] on the server and an interpreter in the client that is able to meet the demands stated above.

Figure 6 shows the user interface that is generated for the BugReport type used in the enterInformation task of the bugReport example in Section 3.4:

```
:: BugReport =
    { application   :: String
    , version       :: Maybe String
    , date          :: Date
    , occursAt      :: BugOccurance
    , severity      :: BugSeverity
    , description   :: Note
    }
:: BugSeverity  = Low | Medium | High | Critical
:: BugOccurance = Startup | Shutdown | Other Note
```

The demands stated above are all applicable to this relatively simple type already. It contains both optional and mandatory parts, it has to adapt dynamically when the Other constructor is chosen and it has a wide variety of input elements that have to be arranged in a pleasing layout. An attentive reader may even spot that different input controls are used to select a constructor in Figure 6 for BugOccurance and BugSeverity. This choice is not specified explicitly, but is decided by a layout heuristic in the interface generation.

4.1 Key Concepts

The iTask system generically provides generic user interfaces through the interplay between two type generic functions. The first one, gVisualize, generates visualizations of values that are rendered by the client. The second one, gUpdate, maps updates in the rendered visualization back to changes in the corresponding values. Before explaining these functions in detail, we first introduce the key concepts underlying their design.

Visualizations Visualizations in the iTask system are a combination of pretty printing and user interface generation. The idea behind this concept is that they are both just ways of presenting values to users, whether it is purely informational or for (interactive) editing purposes. The generic user interface library therefore integrates both in a single generic function. Furthermore, most types of visualizations can be coerced into other types of visualizations. For example: a value visualized as text can be easily coerced to an Html visualization, or vice versa. The library offers functions for such coercions. There are six types of visualizations currently supported as expressed by the following type:

```
:: VisualizationType
    = VEditorDefinition
    | VEditorUpdate
    | VHtmlDisplay
    | VTextDisplay
    | VHtmlLabel
    | VTextLabel
```

And four actual visualizations:

```
:: Visualization
    = TextFragment String
    | HtmlFragment [HtmlTag]
    | TUIFragment TUIDef
    | TUIUpdate TUIUpdate
```

The VHtmlDisplay and VTextDisplay constructors are pretty print visualizations in either plain text or Html. The VHtmlLabel and VTextLabel constructors are summaries of a value in at most one line of text or Html. Labels and display visualizations use the same constructor in the Visualization type. The VEditorDefinition and VEditorUpdate visualizations are explained in the next two subsections.

User Interface Definitions When a value is to be visualized as an editor, it is represented as a high-level definition of a user interface. These TUIDef definitions are delivered in serialized form to a client as part of a TTInteractiveTask node of a task tree. A client can use this definition as a guideline for rendering an actual user interface. The TUIDef type is defined as follows:

```
:: TUIDef
  = TUIButton TUIButton
  | TUINumberField TUINumberField
  | TUITextField TUITextField
  | TUITextArea TUITextArea
  | TUIComboBox TUIComboBox
  | TUICheckBox TUICheckBox
  ...
  | TUIPanel TUIPanel
  ...
:: TUIButton =
  { name        :: String
  , id          :: String
  , text        :: String
  , value       :: String
  , disabled    :: Bool
  , iconCls     :: String
  }
:: TUIPanel =
  { layout      :: String
  , items       :: [TUIDef]
  , buttons     :: [TUIDef]
  ...
  }
```

Components can be simple controls such as buttons described by the `TUIButton` type on line 11, or containers of other components such as the `TUIPanel` type on line 19 that contains two containers for components: One for its main content, and one additional container for action buttons (e.g. "Ok" or "Cancel").

User Interface Updates To enable dynamic user interfaces that adapt without replacing an entire GUI, we need a representation of incremental updates. This is a visualization of the difference between two values expressed as a series of updates to an existing user interface.

```
:: TUIUpdate
  = TUIAdd TUIId UIDef
  | TUIRemove TUIId
  | TUIReplace TUIId UIDef
  | TUISetValue TUIId String
  | TUISetEnabled TUIId Bool
:: TUIId :== String
```

New components can be added, existing ones removed or replaced, values can be set and components can be disabled or enabled. The `TUIId` is a string that uniquely identifies the components in the interface that the operation targets. The one exception to this rule is the `TUIAdd` case, where the `TUIId` references the component *after* which the new component will have to be placed.

User interface updates are computed by a local structural comparison while traversing an old and new datastructure simultaneously. This ensures that only substructures that have changed are being updated.

Data Paths In order to enable updating of values, it is necessary to identify substructures of datastructure. A `DataPath` is a list of integers (`::DataPath :== [Int]`) that are indexes within constructors (of arity > 0) when a datastructure is being traversed. Figure 7 show some example `DataPath`s for a simple binary tree. `DataPath`s are a compact, yet robust identification of substructures within a datastructure.

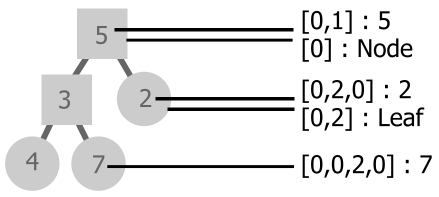


Fig. 7. Data paths for a value of type `::Tree = Node Tree Int Tree | Leaf Int`

Data Masks When a datastructure is edited, it is possible that during this editing, parts of the structure are temporarily in an “invalid” state. For example when an element is added to a list: between the structural extension of the list and the user entering the value of the new element, the list is in a state in which one of its elements has a value, but that is not entered by the user. To indicate which parts of a datastructure have been accessed by a user we use the `DataMask` concept. A `DataMask` is simply a list of all paths that have been accessed by a user (`::DataMask :== [DataPath]`). This additional information is used to enhance usability by treating components that have not been touched by a user different from those that the user has already touched. For example, validation of only those fields in a form that have already been filled out.

4.2 The Big Picture

With the key concepts explained, we can now sketch the big picture of how user interfaces of interactive tasks are handled. This process consists of three main steps:

1. An initial user interface definition (`TUIDef`) representing the current value of a datastructure and its mask is generated by a generic function on the server. This definition is rendered by the client and event handlers are attached to interactive components to notify value changes.
2. When a user changes an interactive component, an encoding of this change and the datapath of the component are sent back to the server and interpreted by another type generic function that updates the datastructure and mask to reflect the change.
3. The updated datastructure is compared to its previous value and if there is a structural difference, a list of `TUIUpdate` is computed and sent back to the client. The client interprets these instructions and modifies the interface accordingly.

In the next section we will explain some of the machinery behind those steps. For reasons of brevity we do not go into implementation details, but explain the key datastructures and type signatures of key functions instead.

4.3 Low Level Machinery

The core machinery of the library consist of two generic functions: `gVisualize` and `gUpdate`. Instances of these functions for concrete types can be automatically derived. Because these functions have been designed favoring pragmatism over elegance, the library exposes them through a set of wrapper functions:

```
//Visualization wrappers (under condition that gVisualize exists for type a)
visualizeAsEditor      :: String DataMask a → ([TUIDef],Bool)
                          | gVisualize{|*|} a
visualizeAsHtmlDisplay :: a → [HtmlTag]
                          | gVisualize{|*|} a
```

```
determineEditorUpdates :: String DataMask DataMask a a → ([TUIUpdate],Bool)
                          | gVisualize{|*|} a
...
//Update wrappers (under condition that gUpdate exists for type a)
updateValueAndMask     :: String String a DataMask *World → (a,DataMask,*World)
                          | gUpdate{|*|} a
...
```

Tasks such as `enterInformation` use the `visualizeAsEditor` wrapper to create the content of a `TTInteractiveTask` node in the task tree. All interactive components are given an identifier derived from their data path within the data structure. This enables the client to send back updates when such a component is updated. When a client sends an event to the server, the `updateValueAndMask` wrapper is used to process the update. Its first two arguments are a string representation of the data path, and a string representation of the update. The last parameter is the unique world. Clean uses uniqueness typing to facilitate stateful functions by threading an abstract `World` value. The main reason that updates are impure, is that it enables impure specializations for specific types. For example when updating a `Maybe Date` from `Nothing` to `Just`, the current date can be set as value. After updating a value and mask, the `determineEditorUpdates` wrapper is used to create task content containing an incremental update for the client GUI.

Although the generic functions are never called directly, and for normal use only *derived* for types, we conclude this section with a brief overview of their type signatures and arguments to give an impression of what goes on under the hood.

```
generic gVisualize a ::
  (VisualizationValue a)
  (VisualizationValue a)
  VSt → ([Visualization], VSt)

:: VisualizationValue a = VValue a DataMask | VBlank
:: VSt =
    { vizType          :: VisualizationType
    , idPrefix         :: String
    , label            :: Maybe String
    , currentPath      :: DataPath
    , useLabels        :: Bool
    , onlyBody         :: Bool
    , optional         :: Bool
    , valid            :: Bool
    }
```

The first two arguments are wrapped values of type `a` with their mask, or an undefined blank. The last argument that is both input and output of `gVisualize` is the visualization state. This state contains all parameters relevant to the visualization and is used to keep track of global properties. The `optional` field in the structure is used to mark parts of editor visualizations as optional. A specialization of `gVisualize` for the `Maybe a` type sets this field to true, and then produces a visualization of type of `a`. When a visualization of an optional value that is `Nothing` needed, there is no value of type `a` available. In that case `VBlank` values are used. The `valid` field of `VSt` is used to validate mandatory fields. It is updated at each interactive element and set to `False` when a non-optional field is not masked. This validation is used to disable completion of a task until its form has been filled out completely.

```
generic gUpdate a        :: a          *USt → (a, *USt)
:: *USt =
    { mode               :: UpdateMode
    , searchPath         :: DataPath
    , currentPath        :: DataPath
    , update             :: String
    , consPath           :: [ConsPos]
    , mask               :: DataMask
    , world              :: *World
    }
:: UpdateMode = UDSearch | UDCreate | UDMask | UDDone
```

The `gUpdate` function traverses a datastructure recursively and at each point transforms the value and state according to one of four modes. In `UDSearch` mode, the `currentPath` path field is compared to the `searchPath` field and `update` is applied when they are equal. The `mode` is then set to `UDDone` and the `mask` field is updated to include the value of `currentPath`.. In `UDDone` mode, the function does nothing and is just an identity function. When a constructor of an algebraic data type is updated to one that has a non-zero arity, the `gUpdate` function needs to be able to produce default values for the substructures of the constructor. It uses its `UDCreate` mode to create these values. In this mode, the `gUpdate` ignores its input value and returns a default value. The last mode is the `UDMask` mode, which adds the paths of all substructures to the `mask` as it traverses the datastructure. This is used to compute a complete mask of a datastructure.

5 Related Work

The iTask system is a workflow management system, and is therefore comparable with other WFMSs. However, unlike many contemporary WFMSs (e.g. YAWL, WebSphere, Staffware, Flower, Bonita), the iTask system does not use a graphical formalism for the specification of workflows, but uses a compact combinator language embedded in a general purpose functional language instead.

Although the iTask system is a WFMS, many web applications can be considered workflow support systems in some way or another. Therefore one could also view the iTask system as a more general framework for (rapid) development of web applications. This makes it comparable with other web development frameworks found in functional languages like WASH/CGI [14]and HAppS [5] in Haskell, XCaml in OCaml, or the frameworks available in dynamic scripting languages like Rails [13] in Ruby or Django [2] in Python. While these frameworks aim to simplify the development of any type of web application, the iTask system will only be suitable for applications that can be sensibly organized around tasks.

The final body of work that may be classified as related is not so much related to the iTask system itself but rather to its new generic web visualization library. Other functional web GUI libraries exist like the WUI combinator library in Curry [4], or the iData toolkit [8] in Clean that powered previous iTask implementations. The new iTask visualization library differs from those libraries in that it makes use of an active Ajax client, in this case built with the ExtJS framework [3]. This gives the generated editors more control over the browser than is possible with plain Html forms, hence enabling the generation of more powerful "desktop-like" user interfaces. However, the iTask client is a single application that interprets instructions generated on the server and is not to be confused with client side application frameworks such as Flapjax [7]. Such frameworks could be used as a replacement for ExtJS in alternative iTask clients.

6 Conclusions

In this paper we have presented a new implementation of the iTask system. This new implementation uses a service-based architecture combined with an active client. This approach enables the generation of more user-friendly interfaces for end-users without compromising the declarative nature of the iTask language.

Although seemingly superficial, improved usability is a crucial aspect of the implementation of the iTask workflow language, because the iTask system generates executable systems solely from a workflow specification and *nothing else*. Hence, the generation quality largely determines the usefulness of the language.

A direct consequence of, and a primary motivation for, this work is that it enables case study and pilot research to validate the effectiveness of the function combinator approach to workflow modeling used by the iTask system in scenarios with real end-users. Not surprisingly, such realistic case studies in the context of supporting disaster response operations are planned for the coming years.

More information, examples and downloads of the iTask system can be found at: http://itasks.cs.ru.nl/.

Acknowledgement

The authors would like to thank Jan Martin Jansen and the anonymous reviewers for their insightful comments, and Steffen Michels for exploring some early ideas about combining server-side generic programming with client-side Javascript.

References

1. Artem Alimarine and Rinus Plasmeijer. A generic programming extension for Clean. In Thomas Arts and Markus Mohnen, editors, *Selected Papers of the 13th International Symposium on the Implementation of Functional Languages, IFL'01*, volume 2312 of *LNCS*, pages 168–186. Springer-Verlag, September 2002.
2. Django framework. `http://www.djangoproject.com/`.

3. ExtJS framework. `http://www.extjs.com/`.
4. Michael Hanus. High-level server side web scripting in Curry. In *Proceedings of the 3rd International Symposium on the Practical Aspects of Declarative Programming, PADL'01*, pages 76–92. Springer-Verlag, 2001.
5. Alexander Jacobson. Haskell application server, 2006. `http://happs.org/`.
6. Pieter Koopman, Rinus Plasmeijer, and Peter Achten. An executable and testable semantics for iTasks. In Sven-Bodo Scholz, editor, *Selected Papers of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08*, 2009. To appear in Springer LNCS.
7. Leo Meyerovich, Arjun Guha, Jacob Baskin, Greg Cooper, Michael Greenberg, Aleks Bromfield, and Shriram Krishnamurthi. Flapjax: A programming language for ajax applications. Tech report, CS-09-04, Brown University, Providence, RI, 04 2009.
8. Rinus Plasmeijer and Peter Achten. iData for the world wide web - Programming interconnected web forms. In *Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS'06*, volume 3945 of *LNCS*, pages 242–258, Fuji Susone, Japan, 24-26, April 2006. Springer Verlag.
9. Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th International Conference on Functional Programming, ICFP'07*, pages 141–152, Freiburg, Germany, 1-3, October 2007. ACM Press.
10. Rinus Plasmeijer, Peter Achten, and Pieter Koopman. An introduction to iTasks: defining interactive work flows for the web. In *Selected Lectures of the 2nd Central European Functional Programming School, CEFP'07*, volume 5161 of *LNCS*, pages 1–40, Cluj-Napoca, Romania, 2008. Springer-Verlag.
11. Rinus Plasmeijer, Peter Achten, Pieter Koopman, Bas Lijnse, and Thomas van Noort. An iTask case study: a conference management system. In *Selected Lectures of the 6th International Summer School on Advanced Functional Programming, AFP'08*, LNCS, Center Parcs "Het Heijderbos", The Netherlands, 19-24, May 2008. Springer-Verlag.
12. Rinus Plasmeijer, Jan Martin Jansen, Pieter Koopman, and Peter Achten. Declarative Ajax and client side evaluation of workflows using iTasks. In *Proceedings of the 10th International Conference on Principles and Practice of Declarative Programming, PPDP'08*, pages 56–66, Valencia, Spain, 15-17, July 2008.
13. Ruby on Rails. `http://rubyonrails.org/`.
14. Peter Thiemann. WASH/CGI: server-side web scripting with sessions and typed, compositional forms. In Shriram Krishnamurthi and Raghu Ramakrishnan, editors, *Proceedings of the 4th International Symposium on the Practical Aspects of Declarative Programming, PADL'02*, volume 2257 of *LNCS*, pages 192–208, Portland, OR, USA, 19-20, January 2002. Springer-Verlag.