Using a Functional Language as Embedding Modeling Language for Web-Based Workflow Applications

Rinus Plasmeijer[1], Jan Martin Jansen[2], Pieter Koopman[1], Peter Achten[1]

[1] Institute for Computing and Information Sciences (ICIS),
Radboud University Nijmegen, the Netherlands,
[2] Faculty of Military Sciences,
Netherlands Defence Academy, Den Helder, the Netherlands
jm.jansen.04@nlda.nl, {rinus, pieter, peter88}@cs.ru.nl

Abstract. Workflow management systems guide and monitor tasks performed by humans and computers. Workflow specifications are usually expressed in special purpose (graphical) formalisms. These models are concise and made rapidly. These formalisms commonly have as disadvantage that they have limited expressive power, handle only rather static workflows, do not handle intricate data dependencies, and cannot easily adapt to the current situation. Furthermore, workflow specification tools mostly generate a software infrastructure that needs to be extended with custom crafted code. To overcome these problems, we *entirely embed* a workflow modeling language in a modern general purpose *functional* language, and *generate* a complete workflow application. We have developed the iTask prototype system in the pure and lazy language Clean. An iTask workflow specification can use the expressive power and strong type facilities of Clean. Workflows can be higher-order and adapt their behavior based on the available data. The generated application is web-based and runs on both server and clients.

1 Introduction

Workflow Management Systems (WFMS) are computer applications that coordinate, generate, and monitor tasks performed by human workers and computers. Workflow *models* play a dominant role in WFMSs: the work that needs to be done to achieve a certain goal is specified in such a model as a structured and ordered collection of tasks that are assigned to available resources at run-time. In most WFMSs, a workflow model is used as input to generate a framework, i.e. a *partial* workflow *application*. Substantial coding is required to complete the workflow application. For example, most workflow models only deal with the flow of control of the application. All code with respect to manipulating the data in the workflow application has to be implemented separately. In this paper we advocate that a workflow model actually *can be* a computer program: an entire workflow application can and should be generated from a workflow model.

Contemporary WFMSs use special purpose (mostly graphical) modeling languages. Graphical formalisms are easier to communicate to domain experts than textual ones. Domain experts are knowledgeable about the work to be modeled, but often lack programming experience or formal training. Special purpose modeling languages provide workflow engineers with a concise formalism that enables the rapid development of a workflow framework. Unfortunately, these formalisms suffer from a number of disadvantages when compared with textual ones. First, *recursive definitions* are commonly inexpressible, and there are only limited ways to make *abstractions*. Second, workflow models usually only describe the *flow of control*. Data involved in the workflow is mostly maintained in databases and is extracted or inserted when needed (see the 'Data Interaction – Task to Task' workflow data pattern (8) by Russell *et al*[16]). As a consequence, workflow models cannot easily use this data to parameterize the flow of work. The workflow is more or less pre-described and cannot be dynamically adapted. Third, these dedicated languages usually offer a fixed set of *workflow patterns* [17]. However, in the real world work can be arranged in many ways. If it does not fit in a (combination of) pattern(s), then the workflow modeling language probably cannot cope with it either. Fourth, and related, is the fact that special purpose languages cannot express functionality that is not directly related to the main purpose of the language. To overcome this limitation, one either extends the special language or interfaces with code written in other formalisms. In both cases one is better off with a well designed general purpose language.

For the above reasons, we advocate to use a textual *programming language* as a workflow modeling language. This allows us to address all computational concerns in a workflow model and provides us with general recursion. We use a *functional* language, because they offer a lot of expressive power in terms of modeling domains, use of powerful types, and functional abstraction. We use the *pure* and *lazy* functional programming language Clean, which is a state-of-art language that offers fast compiler technology and *generic programming features* [1] which are paramount for generating systems from models in a type-safe way. Clean is freely available at `http://clean.cs.ru.nl/`.

To verify our claim, we have developed a prototype workflow modeling language called iTask [12, 13]. The iTask system is a *combinator library*. In the functional programming community, combinators are a proven method to embed domain specific languages within a functional host language: application patterns are captured with *combinator functions*, and the application domain is defined by means of the expressive type system, using algebraic, record, and function types. Workflows modeled in iTask result in complete workflow applications that run on the web distributed over server and client side [15].

The remainder of this paper is organized as follows. We present iTask in Sect. 2, and demonstrate the advantages of using a functional programming language as workflow modeling language. We continue with a larger example in Sect. 3. We discuss the major design decisions in Sect. 4. Related work is discussed in Sect. 5. We conclude in Sect. 6.

2 Overview of the iTask system

In this section we give an overview of the iTask system. We start with basic iTasks in Sect. 2.1. We present only a small subset of the available combinators in Sect. 2.2. In Sect. 2.3 we show how embedding iTasks in a general purpose language makes the system extensible and adaptable.

2.1 Basic iTasks

An iTask is a unit of work to be performed by a worker or computer. An iTask can be in different states. It can be: non-active (does not exist yet), active (someone is still working on it), or finished. Clean is a statically typed language, everything has a type. A type can be regarded as a model and a value of that type as an instance of that model. An iTask has the following opaque, parameterized type:

```
:: Task a
```

The type parameter `a` is the type of the value that is delivered by a task. The Clean compiler infers and checks the concrete type of any specified task.

The iTask library offers several functions for creating basic units of work: a basic task. For instance, the function `editTask` takes a `label` of type `String` and an initial `value` of some type `a` and creates a `Task a`: a form in a web page in which the worker can edit this initial `value`. Its type is:

```
editTask :: String a → Task a | iData a
```

The function `editTask` is very powerful. It creates an editor for *any* first-order concrete type and handles *all* changes. A worker can change the value as often as she likes but she cannot alter its type. When the `label` button is pressed, the `editTask` is finished and the final value is delivered as result.

`EditTask` is not a polymorphic function (i.e. *one* function which works for *any* type), but it is overloaded. For each type it is applied on, a special version is constructed. One can regard `editTask` as a kind of *type driven* or *model driven* function. The precise working of the function depends on the concrete type (model) on which it is applied. This is *statically* determined. The type dependent behavior is inductively defined on a small number of generic, type driven functions [1], which is reflected in the type context restriction (`| iData a`) in the type definition of `editTask`. A small, but complete iTask workflow application is:

```
module SmallButCompleteExample                                              1.
import iTasks                                                               2.

Start        :: *World → *World                                             3.
Start world  = startEngine [addFlow ("simple", simpleEditor)] world         4.

simpleEditor :: Task Int                                                    5.
simpleEditor = editTask "Done" createDefault                                6.
```

The application imports the iTask library (line 2). Execution begins with the `Start` function. It calls the iTask engine, and adds the workflow `simpleEditor` to the

workflow list that can be invoked by workers. `SimpleEditor` (line 6) only consists of one invocation of `editTask`. Eventually, it produces an `Int` value, indicated by the type definition (line 5). The button labeled `Done` finishes the task. The library function `createDefault` is used as initial value. It creates a default value for *any* type (hence it is also a generic function). The default value for type `Int` is 0.

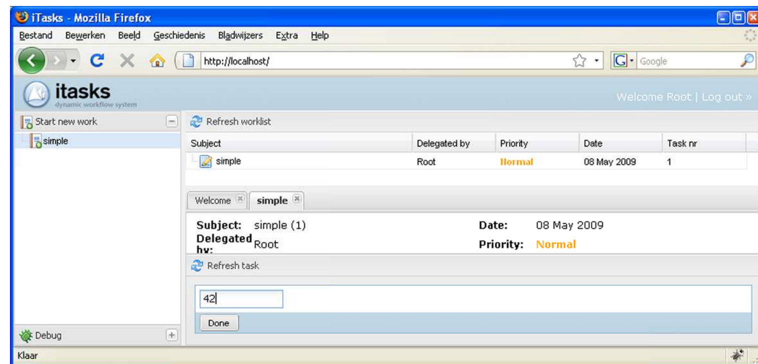

Fig. 1. Generated webpage for the `simpleEditor` example.

The generated web application is shown in Fig. 1. After logging in, the application resembles a regular e-mail application. The names of the tasks that the worker needs to perform are presented in the *task list* displayed in the right upper pane. This pane can be compared with the list of incoming e-mails. When the worker clicks on a task in the task list, the current state of it is displayed in the right lower *task pane*. Tasks can be selected from the task list in any order. The iTask toolkit automatically keeps track of all progress, even if the user quits the system. When a task is finished, it is removed from the task list. Workers can start new workflows, by selecting them in the left *workflow pane*. In our example there is only one option. In general arbitrarily many workflows can be started, one can assign different workflow options for different types of workers, and the options to choose from can be controlled dynamically. The task list is updated when new tasks are generated, either on her own initiative, or because they have been delegated to her. The entire interface is generated completely and automatically from the sole specification shown above.

In the above example the worker can only enter `Integer` values. Suppose we want a similar workflow for a custom model type, say *person*. We define the necessary domain types (`Person` and `Gender`), derive framework code for these model types, and change the type of `simpleEditor` to `Task Person` (Fig. 2).

Hence, the form that is created depends on the type of the value that the editor should return. For any (user defined) type a standard form can be automatically generated in this way. The details of how a form is actually represented can be fine-tuned in a separate `CSS` file. A programmer can specialize the form generation for a certain type if a completely different view is wanted. One is

```
:: Person = { firstName   :: String
            , surName     :: String
            , dateOfBirth :: HtmlDate
            , gender      :: Gender
            }
:: Gender = Male | Female
derive iData Person, Gender
```


Fig. 2. A standard form editor generated for type `Person`.

not restricted to use standard browser forms. It is e.g. possible to use a drawing plug-in as editor for making values of type, say `Picture` [7].

The function `editTask` is one of several basic tasks. Examples of other basic task functions are: obtaining all users of the system (if necessary grouped by their role); tasks that return at a predefined moment in time or after an amount of time; tasks that can store information in or read information from a database.

2.2 Basic iTask Combinators

New tasks can be composed out of (basic) tasks by using *combinator* functions. As said before, work can be organized in many ways. The expressive power of the host language allows us to cover all common workflow patterns listed in [17], and many more, using only relatively few combinators. Here we discuss a few of them and show their usage.

Sequential composition of tasks can be realized using *monadic* [20] combinator functions for binding (>>=), called *bind*, and emitting (`return`) values:

```
(>>=) infixl 1 :: (Task a) (a → Task b) → Task b | iData b
return         :: a                     → Task a | iData a
```

In contrast to most workflow specification languages, in the iTask system information is passed explicitly from one task to another. The first task (of type `Task a`) is activated first and when it finishes, >>= takes care that its result (of type `a`) is passed to the second argument (a *function* of type `a → Task b`). This function can inspect the result produced by the previous task and react on it. When the second task is finished it produces a value of type `b`. The result of bind is a task of type `Task b`. Note that *t* >>= *f* integrates *computation* and *sequential ordering* in a single pattern. This is hard to specify in a graphical modeling language. The `return` combinator lifts any value (of some type `a`) to a (`Task a`) that yields that value. Before we give an example of sequential composition, we introduce one of the *choice* operators: `chooseTask`.

```
chooseTask :: [HtmlTag] [(String,Task a)] → Task a | iData a
```

A worker is offered a choice out of a list of tasks with `chooseTask`. Each task is identified with a label (of type `String`) which is used to display the options to the worker. When one of the options is chosen, the corresponding task is selected

and performed. The additional prompt argument (of type `[HtmlTag]`) can be used to explain to the worker what the intention is. Any Html-code can be used.

```
chooseUser :: Task (UserId,String)                                          1.
chooseUser                                                                  2.
=              getUsers                                                     3.
  >>= λusers → chooseTask  [Text "Select worker who has to do the job:"]    4.
                           [(name,return user) \\ user=:(_,name) ← users]   5.
```

The example task `chooseUser` uses several of the basic tasks and combinators introduced so far. The basic task `getUsers` returns the list of known users. These are passed to `chooseTask` to allow the worker to select one known user. The *list comprehension* in line 5 is well known functional idiom for manipulating lists.

Tasks can be *assigned* to workers. This is done with the `@:` combinator:

```
(@:) infix 3 :: UserId (String, Task a) → Task a | iData a

delegateTask :: String (a → Task a) a → Task a | iData a
delegateTask taskname taskf val
=   chooseUser >>= λ(user,_) → user@:(taskname, taskf val)
```

The `@:` operator is a basic combinator with which any task can be assigned to a worker (type `UserId`). The label (of type `String`) gives a name to the task (displayed in the task list of this specific user). `DelegateTask` shows its usage: first a worker is chosen with `chooseUser` and this worker is assigned the given task.

Workers can be *prompted* about their progress with `?>>`:

```
(?>>) infixr 5 :: [HtmlTag] (Task a) → Task a | iData a

taskToDelegate :: a → Task a | iData a
taskToDelegate value
=   [Text "Would you be so kind to fill in the following form:"]
    ?>> editTask "TaskDone" value
```

The `?>>` combinator can be used to add a prompt to any task. The prompt (any Html-code can be used here again) is displayed as long as the task is not finished. This facility is used in `taskToDelegate` to make a polite version of `editTask`.

Notice that all definitions above can be applied for any task of any type (read: model). In `concreteDelegate` below we apply `delegateTask` to a general task for filling a form (`taskToDelegate`) given some initial value (`createDefault`). As was the case with a simple form editor, the type specified here completely determines the actual form to be filled in by the chosen worker:

```
concreteDelegate :: Task Person
concreteDelegate = delegateTask "person" taskToDelegate createDefault
```

Finally, iTask specifications can use all features of Clean including *recursion*.

```
recursiveDelegate:: String (a → Task a) a →  Task a  | iData a                         1.
recursiveDelegate taskname taskf val                                                   2.
=                 delegateTask taskname taskf val                                      3.
    >>= λresult → chooseTask [Text "Result:", toHtml result, Text "Approved ?"]        4.
```

```
                [ ("Yes",return result)                                          5.
                , ("No", recursiveDelegate taskname taskf result)                6.
                ]                                                                7.

concreteRecDelegate :: Task Person                                               8.
concreteRecDelegate = recursiveDelegate "person" taskToDelegate createDefault    9.
```

A variant of `delegateTask` is given in `recursiveDelegate`. The worker who delegates the task receives the result back (in `result`) for examination. It is shown in the prompt of `chooseTask`: `toHtml` (line 4) displays any result of any type. If the worker chooses not to approve the result, `recursiveDelegate` is called recursively and the delegation of the work starts all over, but now with the latest result as starting point (line 6). Again this recursive version can be used for any task and any type: it is turned into a `Person` workflow by `concreteRecDelegate` (line 8).

2.3 The Expressive Power of the Combinators

In [17] an inventory is made of the workflow patterns offered by commercial WFMSs. This collection is rather large. The reason for this is that the underlying workflow languages generally do not offer the right abstraction mechanism for defining new patterns. In this section we show how iTask makes use of the functional host language to capture new patterns concisely.

Two commonly available workflow patterns are `orTasks` and `andTasks`. Both work on a collection of tasks but have different termination conditions: `orTasks` finishes as soon as one of its subtasks finishes, `andTasks` finishes as soon as all subtasks are finished. We can capture this common behavior with a more general combinator, `parallel`, that can be used to define `orTasks` and `andTasks`:

```
parallel :: String ([a] → Bool) (Bool → [a] → b) [(String, Task a)] → Task b
         | iData a & iData b

orTasks  = parallel "orTasks"  (not o isEmpty) (const hd)
andTasks = parallel "andTasks" (const False)   (const id)
```

The `parallel` combinator is given a predicate (of type `[a] → Bool`) which determines when to stop. The predicate is applied on the list of values of all completed subtasks. As soon as the predicate holds, all unfinished subtasks are stopped (even if other workers are working on it) and the results of the finished tasks are collected and converted (by the function of type `Bool → [a] → b`) to the type `b` demanded by the application. The `Bool` argument is true iff the predicate was satisfied. `Parallel` also terminates when all subtasks have terminated, but the predicate is still invalid. With `parallel`, `orTasks` and `andTasks` are easily expressed, as shown above. (The `const` function ignores the boolean argument and applies its argument function to the list of results.) Many other imaginable and useful patterns can be defined in a similar way, e.g. when one wants to stop as soon as enough information has been produced by the finished tasks (see also Sec. 3), or as soon as someone has given up. Such user defined stop criteria cannot be defined in most commercial workflow languages.

Another useful pattern is the ability to cancel a task (even when performed by someone else), which can concisely be expressed with `orTasks` as shown in the function `cancelable`.

```
:: Maybe a = Just a | Nothing

cancelable :: (Task a) → Task (Maybe a) | iData a
cancelable task
= orTasks [("Cancel",editTask "Cancel" Void >>= λ_ → return Nothing )
          ,("Normal",task >>= λresult → return (Just result))
          ]

cancelableDelegate = cancelable concreteRecDelegate
```

The `Void` type has no visualization, hence `editTask` only shows a button labeled `Cancel`. `Nothing` is returned when `Cancel` is pressed.

The iTask library has several of these general purpose combinators, but there is no room here to discuss them all. There are combinators for workflow process creation and handling, thread handling, exception handling, and combinators which enable the change of work under execution. Compared to the set of well-known workflow patterns, we have far less combinators yet we can express a lot more different work situations. The reason is that with Swiss-army-knife combinators such as `parallel` and the expressive power of the host language we can construct not only many well-known workflow patterns, but also new variants, as explained above. Furthermore, the system is open ended: the programmer can add new combinators when the basic collection is insufficient.

3 Order Booking Example

To demonstrate the expressive power of iTask, we present an *order booking* example. The code presented below is a complete, executable, iTask workflow. The workflow has a recursive structure and monitors intermediate results in a parallel and-task. This case study is hard to express in traditional workflow systems. The overall structure contains the following steps (see `getSupplies` below): first, an inventory is made to determine the required amount of goods (`getAmount`) (e.g. vaccines for a new influenza virus); second, suppliers are asked in parallel how much they can supply (`inviteOffers`); third, as soon as sufficient goods can be ordered, these orders are booked at the respective suppliers (`placeOrders`). A `Supplier` is a pair (`UserId`,`String`), and `Amount` is a non-negative `Int`.

```
getSupplies :: Task [Void]                                    1.
getSupplies = getAmount >>= inviteOffers >>= placeOrders      2.
```

Determining the required amount of goods also proceeds in two steps: first, the institute enquires other institutes *recursively* in parallel (using the `andTasks` combinator) how many goods they need (lines 6-9). This means that each of these institutes can ask other institutes for the same thing, and so on. Note that an institute can decide to select no other institutes. In that case the recursion

stops. Second, given this amount, the institute can alter this number (line 11). Also, `chooseUsers` (line 6) is a variant of `chooseUser` (Sect. 2.2) that uses *multiple choice* and yields a list of workers. The operator `<+` converts its second argument to `String` and concatenates it to its first argument.

```
getAmount :: Task Amount                                                3.
getAmount =         [Text "Ask other institutes"] ?>> chooseUsers       4.
    >>= λinsts  →  andTasks [( "Request for " <+ name                   5.
                              , uid @: ("Amount request", getAmount)    6.
                              ) \\ (uid,name) ← insts]                  7.
    >>= λothers →  [Text "Enter the required amount"]                   8.
                   ?>> editTask "Done" (sum others)                     9.
```

Once the amount of goods is established, the workflow can continue by inviting offers from a collection of candidate suppliers. This collection is determined first (line 14). Each supplier can provide an amount (line 18). This is again done in parallel (line 15-20). The termination criterium is the `enough` predicate which is satisfied as soon as the sum of provided offers exceeds the requested amount (line 22). The canonization function `maximum` is discussed below. Hence, the result of this workflow task is a list of offers. Each offer is a pair of a supplier and the amount of goods that it offers to deliver.

```
inviteOffers :: Amount → Task [(Supplier,Amount)]                       10.
inviteOffers needed                                                     11.
=              [Text "Choose candidate suppliers"] ?>> chooseUsers      12.
  >>= λsups → parallel "Supplier_requests" enough (maximum needed)      13.
               [("Request for " <+ name                                 14.
                ,uid @: ("Order request"                                15.
                        ,prompt ?>> editTask "Done" needed >>= λa → return (sup,a) 16.
                )       )                                               17.
               \\ sup=:(uid,name) ← sups                                18.
               ]                                                        19.
where enough as = sum (map snd as) >= needed                            20.
      prompt    = [Text "Request for delivery, how much can you deliver?"] 21.
```

The total number of offered goods can differ from the required number of goods. The function `maximum` makes sure that not too many goods are ordered. This is an easy exercise in functional programming:

```
maximum :: Amount Bool [(Supplier,Amount)] → [(Supplier,Amount)]        22.
maximum needed enough offers                                            23.
| not enough                = offers                                    24.
| otherwise                 = [(supplier,exact) : less]                 25.
where [(supplier,_) : less] = sortBy (λ(_,a1) (_,a2) → a1 > a2) offers  26.
      exact                 = needed - sum (map snd less)               27.
```

With the correct list of offerings, we can place an order for each supplier. This can be expressed directly with `andTasks`:

```
placeOrders :: [(Supplier,Amount)] → Task [Void]                        28.
placeOrders offers                                                      29.
=   andTasks [("Order for " <+ name,                                    30.
```

```
          uid @: ("Order request for " <+ name                               31.
                 ,[Text ("Please deliver " <+ a)] ?>> editTask "Done" Void)  32.
                 )                                                           33.
        \\ ((uid,name),a) ← offers ]                                         34.
```

4 Modeling by Abstracting from Details

An iTask workflow specification is modeled in a functional language. This means that the host language needs to support abstraction from (almost) all annoying details. On the other hand, all information has to be provided somehow because from this single source specification a complete real working distributed web application has to be generated. How has this been realized?

Defining General Workflow Schemes. By using polymorphic, overloaded, and generic (type driven) functions, one can specify custom workflow schemes for any frequently occurring work situation that works for *any* concrete task of *any* type. The type system ensures that everything is type correct. To get a working application, a scheme somewhere has to be applied to a concrete model (i.e. type). This is used by the compiler to generate specialized functions that handle the low level details.

Abstracting from Form Views and Form Handling. One does not have to worry about the user interface as a whole. It is defined separately, handled by the client and it can be fine-tuned if desired without affecting the workflow specification. Using generic programming techniques, an editor can be generated for any concrete type specified by the programmer, the information is displayed in a web page and can be interactively modified. Any change made in the form is handled automatically by the application. The input given interactively is type checked; it is impossible to create ill-typed values. It is also possible to specify user-defined predicates over the input which consistency is checked at run-time.

Abstracting from specific Html Output. Sometimes it is necessary to show additional information to the worker, e.g. for prompting and feedback. Abstracting from this is not always completely possible. However, with the function `toHtml` any value of any type can be converted to Html-code (Sec. 2). Furthermore, it is possible to abstract from a concrete prompt or feedback by defining functions for it, which can be given as argument to `?>>` (Sec. 2).

Abstracting from Layout. Lay-out details of the output generated can be specified separately as is common in web applications. The dynamic behavior of the iTask system requires additional run-time facilities for controlling the lay-out.

```
:: TaskCombination
= TTSplit [HtmlTag] | TTVertical| TTHorizontal | TTCustom ([[HtmlTag]] → [HtmlTag])

myCancelable task = cancelable task <<@ TTVertical
```

To influence the layout of Html-code generated by subtasks, one can use the (overloaded) tuning operator `<<@`. One can place the Html-code of each subtask in a separate worktab (`TTSplit`), below each other(`TTVertical`), next to each

other (`TTHorizontal`), or give a user defined recipe how to combine it (`TTCustum`). `MyCancelable` gives an example of its usage. `<<@` can be used to fine tune a system, but the application also works without it.

Abstracting from Storage. In the iTask system information is passed explicitly from one task to another. This may involve different workers. Between the events all information has to be stored somewhere. For a web-application there are many possibilities, e.g. in a file, in a database, on the client. Furthermore, we need to remember the progress of every worker. Again, by using generic programming techniques, all this information is stored fully automatically. With the overloaded tuning operator `<<@` the workflow programmer can decide *where* and in what *format* the information is stored (text, binary).

An iTask application also needs access to information stored in standard information systems. We do not have to bother workflow programmers with something low level as SQL queries. We can systematically convert an information model defined in e.g. ORM to Clean data type definitions. This enables the automatic conversion between values of these types and the corresponding values stored in a relational database [9].

Abstracting from the Web Architecture. The architecture (Fig. 3) of iTask applications is representative for Ajax [4] based web applications. Initially a web page is generated as shown in Fig. 1. Whenever a worker clicks on something, an

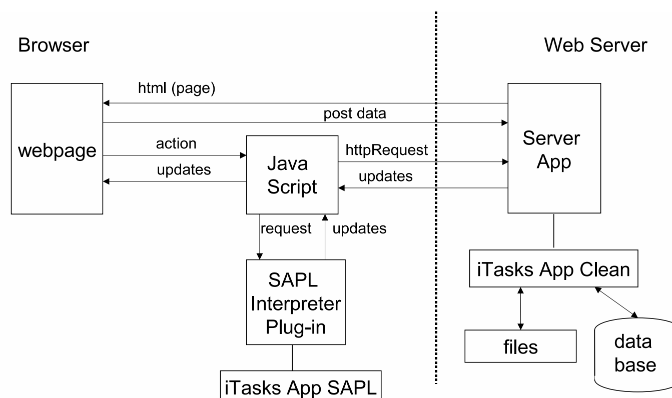

Fig. 3. The architecture of an iTask application

asynchronous Ajax request is sent to the iTask application to handle it. It reacts by sending an update of those parts of the page that have to be changed. All web handling is done automatically. Although one is defining a multi-user web application, one does not need any knowledge about the underlying architecture.

One of the most interesting aspects of the architecture is that it is possible to execute, in principle, *any* iTask on either the server or on a client [15]. For example, a function like `concreteDelegate` (Sect. 2.2) is executed on the server by default. It can be executed on the client by using the tuning operator `<<@` and the

`OnClient` pragma: `concreteDelegate <<@ OnClient`. The `<<@` operator accepts *any* task. However, not all tasks can be executed (completely) on the client (e.g. tasks that access a database). This is detected at run-time and the system automatically migrates such a task back to the server.

To implement this, the Clean compiler generates *two* executable instances from a single source. An executable iTask system runs on the server, while an interpreted version runs on every client. Both implementations are efficient. The Clean compiler is well-known for the excellent code it generates which is comparable to C. In the browser we use the Sapl-interpreter [6] which is currently one of the fastest interpreters for a functional language. One JavaScript function decides for each user action whether it is handled on the client or on the server.

Abstracting from the Evaluation Order. The iTask combinators define the order in which tasks can be executed: tasks can be performed sequentially, in parallel, distributively (on clients), and there even is strong support for exception handling. Notice that iTasks are just plain Clean applications (there is no special interpreter for iTask applications or something like that). A functional language like Clean evaluates expressions in a fixed way: lazily (normal order). Clean does not support parallel evaluation, distributed evaluation, nor exception handling. It is very remarkable that the wild scala of evaluation orders offered by the iTask library can nevertheless be embedded in the host language. This has been achieved by making clever use of generic techniques for the automatic storage of the state of the iTask application in combination with re-evaluation of (part of) the application [12]. This allows us to mimic any evaluation strategy without the need to make any change or extension in the host language.

5 Related Work

The WebWorkFlow project [5] shares our point of view that a workflow specification is regarded as a web application. WebWorkFlow is an object oriented workflow modeling language that is embedded in WebDSL [19], a domain specific language for developing web applications. In WebWorkFlow, *workflow objects* accumulate the progress made in a workflow. *Workflow procedures* define the actual workflow. Their specification is broken down into *clauses* that individually control *who* can perform *when*, what the *view* is, what should be *do*ne when the workflow procedure is applied, and what further workflow procedures should be *process*ed afterwards. Like in iTask, one can derive a GUI from a workflow object. The main difference is that iTask is embedded in a functional language, but this has significant consequences: iTask supports higher-order functions in both the data models and the workflow specifications; arbitrary recursive workflows can be defined (WebWorkFlow is restricted to tail recursion and recursion on simpler structures); reasoning about the evaluation of an iTask program is reasoning about the combinators instead of the collection of clauses.

Brambilla *et al*[3] enrich a domain model (specified as UML entities) with a workflow model (specified as BPMN) by modeling the workflow activities as additional UML entities and use OCL to capture the constraints imposed by the

workflow. The similarity with iTask is to model the problem domain separately. However, in iTask a workflow is a function that can manipulate the model values in a natural way, which enables us to express functional properties seamlessly (Sect. 3). This connection is ignored in [3] and can only be done ad-hoc.

Pešić and van der Aalst [11] base an entire formalism, ConDec, on linear temporal logic (LTL) constraints. Frequently occurring constraint patterns are represented graphically. This approach has resulted in the DECLARE tool [10]. In iTask a workflow can use the rich facilities of the host language for computations and data declarations – such facilities are currently absent in DECLARE.

Andersson *et al*[2] distinguish high level *business models* (value transfers between *agents*), low level *process models* (workflows in BPMN), and medium level *activity dependency models* (activities for value transfers of business models). Activities are *value transfer*, *assigning* an agent to a value transfer, value *production*, and *coordination* of mutual value transfers and activities. Activities are modeled as nodes in a directed graph. The edges relate activities in a way similar to [3] and [11]: they capture the workflow, but now at a conceptual level. A *conformance relation* is specified between a process model and an activity dependency model. Currently, there is no tool support for their approach. The activity dependency models provide a declarative foundation to bridge the gap between business models and process models. One of the goals of the iTask project is to provide a formalism that has sufficient abstraction to accomodate both business models and process models.

Vanderfeesten *et al*[18] have been inspired by the *Bill-of-Material* concept from manufacturing, recasted as *Product Data Model* (PDM). A PDM is a directed graph. Nodes are product data items, and arcs connect at least one node to one target node, using a functional style computation to determine the value of the target. A tool can inspect which product data items are available, and hence, which arcs can be computed to produce next candidate nodes. This allows for flexible scheduling of tasks. Similarities with the iTask approach are the focus on tasks that yield a data item and the functional connection from source nodes to target node. We expect that we can handle PDM in a similar way in iTask. iTask adds to such an approach strong typing of product data items (and hence type correct assembly) as well as the functions that connect these data items.

6 Conclusions and Future Work

The iTask system demonstrates that a high level general purpose functional language such as Clean (or Haskell which is also supported by the Clean compiler) is very suited to embed a special purpose modeling language. A library provides the domain specific constructs and the specification language inherits the power and advantages of the embedding language. Because only pure functions can be used, one is forced to model in a mathematical way. Strong typing prevents modeling mistakes. The use of generic (type driven) functions enable the use of types as models. Powerful abstraction mechanisms allow to abstract from annoying details such that one can concentrate on modeling. With relatively

few combinators all common workflow patterns are supported, but many more complex workflow situations can be expressed. iTask workflows are dynamic: the flow can depend on the outcome of other tasks. From the specification we are able to generate a real working, distributed evaluated, web enabled, multi-user workflow system. Any task can be shifted from the server to the client. The whole system is generated from one source: a concise iTask specification defined in Clean. Reasoning about the behavior of the system is relatively easy. The semantics of the iTask system and the properties it has are treated in [8].

We have tested the system with larger examples, such as a Conference Management System [14]. We are extending our prototype to a full system which can be applied in industrial environments. In collaboration with the Netherlands Ministry of Defense we are investigating the suitability of the iTask system for handling operational planning and crisis management scenarios. To support these kind of complex dynamic applications we are currently adding the ability to adapt and/or extend running workflows on the fly.

References

1. A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. In T. Arts and M. 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, Sept. 2002.
2. B. Andersson, M. Bergholtz, and A. Edirisuriya. A Declarative Foundation of Process Models. In O. Pastor and J. Cunha, editors, *Proceedings 17 Int'l Conference on Advanced Information Systems Engineering, CAiSE 2005*, volume 3520 of *LNCS*, pages 233–247. Springer-Verlag, 2005.
3. M. Brambilla, J. Cabot, and S. Cornai. Automatic Generation of Workflow-Extended Domain Models. In G. Engels, B. Opdyke, D. Schmidt, and F. Weil, editors, *Proceedings Model Driven Engineering Languages and Systems, 10th Intl' Symposium, MoDELS 2007*, volume 4735 of *LNCS*, pages 375–389. Springer-Verlag, 2007.
4. J. Garrett. Ajax: a new approach to web applications, 18, Feb. 2005.
5. Z. Hemel, R. Verhaaf, and E. Visser. WebWorkFlow: an object-oriented workflow modeling language for web applications. In K. Czarnecki, I. Ober, J. Bruel, A. Uhl, and M. Völter, editors, *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems, MoDELS'08*, volume 5301 of *LNCS*, pages 113–127. Springer-Verlag, 2008.
6. J. Jansen, P. Koopman, and R. Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In H. Nilsson, editor, *Selected Papers of the 7th Symposium on Trends in Functional Programming, TFP'06*, volume 7, pages 73–90, Nottingham, UK, 2006. Intellect Books.
7. J. Jansen, P. Koopman, and R. Plasmeijer. iEditors: extending iTask with interactive plug-ins. In S.-B. Scholz, editor, *Proceedings of the 20th International Symposium on the Implementation and Application of Functional Languages, IFL'08*, pages 170–186, Hertfordshire, UK, 10-12, Sept. 2008. University of Hertfordshire.
8. P. Koopman, R. Plasmeijer, and P. Achten. An executable and testable semantics for iTasks. In S.-B. Scholz, editor, *Proceedings of the 20th International Symposium*

on the Implementation and Application of Functional Languages, IFL'08, pages 53–64, Hertfordshire, UK, 10-12, Sept. 2008. University of Hertfordshire.

9. B. Lijnse. Between types and tables - Generic mapping between relational databases and data structures in Clean. Master's thesis, Institute for Computing and Information Sciences, Radboud University Nijmegen, The Netherlands, July 2008. Number 590.
10. M. Pešić. *Constraint-based workflow management systems: shifting control to users*. PhD thesis, Technical University Eindhoven, 8, Oct. 2008.
11. M. Pešić and W. van der Aalst. A declarative approach for flexible business processes management. In J. Eder and S. Dustdar, editors, *Proceedings of the 1st Business Process Management Workshop on Dynamic Process Management, DPM'06*, volume 4103 of *LNCS*, pages 169–180. Springer-Verlag, 2006.
12. R. Plasmeijer, P. Achten, and P. 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, Oct. 2007. ACM Press.
13. R. Plasmeijer, P. Achten, and P. 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.
14. R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, and T. 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.
15. R. Plasmeijer, J. Jansen, P. Koopman, and P. 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*, Valencia, Spain, 15-17, July 2008.
16. N. Russell, A. ter Hofstede, D. Edmond, and W. van der Aalst. Workflow resource patterns: identification, representation and tool support. Technical report, FIT-TR-2004-01, Queensland University of Technology, Brisbane, Australia, 2004.
17. W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. QUT technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, Australia, 2002.
18. I. Vanderfeesten, H. Reijers, and W. van der Aalst. Product based workflow support: dynamic workflow execution. In Z. Bellahsène and M. Léonard, editors, *Proceedings of the 20th International Conference on Advanced Information Systems Engineering, CAiSE'08*, volume 5074 of *LNCS*, pages 571–574, Montpellier, France, 2008. Springer-Verlag.
19. E. Visser. WebDSL: a case study in domain-specific language engineering. In R. Lämmel, J. Visser, and J. a. Saraiva, editors, *Selected Lectures of the 2nd International Summer School on Generative and Transformational Techniques in Software Engineering, GTTSE'07*, volume 5235 of *LNCS*, pages 291–373, Braga, Portugal, 2-7, July 2007.
20. P. Wadler. Comprehending monads. In *Proceedings of the 6th Conference on Lisp and Functional Programming, LFP'90*, pages 61–77, Nice, France, 1990.