Embedding a Web-Based Workflow Management System in a Functional Language

EXPERIENCE PAPER

Jan Martin Jansen

Faculty of Military Sciences, Netherlands Defence Academy, Den Helder, the Netherlands

Rinus Plasmeijer, Pieter Koopman, and Peter Achten

Institute for Computing and Information Sciences (ICIS), Radboud University Nijmegen, the Netherlands

Abstract

Workflow management systems guide and monitor tasks performed by humans and computers. The workflow specifications are usually expressed in special purpose (graphical) formalisms. These formalisms impose severe restrictions on what can be expressed. Modern workflow management systems should handle intricate data dependencies, offer a web-based interface, and should adapt to dynamically changing situations, all based on a sound formalism. To address these challenges, we have developed the iTask system, which is a novel workflow management system. We *entirely embed* the iTask specification language in a modern general purpose *functional* language, and *generate* a complete workflow application. In this paper we report our experiences in developing the iTask system. It not only inherits state-of-the-art programming language concepts such as generic programming and a hybrid static/dynamic type system from the host language Clean, but also offers a number of novel concepts to generate complex, real-world, multi-user, web based workflow applications.

1 Introduction

Workflow Management Systems (WFMS) are computer applications that coordinate, generate, and monitor tasks performed by human workers and computers. Workflow *specification* plays a dominant role in WFMSs: the work that needs to be done to achieve a certain goal is specified as a structured and ordered collection of tasks that are assigned to available resources at run-time. In

many WFMSs, the workflow specification only determines the framework for the workflow application, i.e. a *partial* workflow *application*. In other WFMSs one has to provide much details in the workflow specification. In both approaches substantial coding is required to complete the workflow application. In general, this results in complex distributed, multi-user and heterogeneous applications that are hard to maintain.

In this paper, we report on our experience in designing, building, and deploying the iTask system [12], which is a novel WFMS based on state-of-the-art programming language concepts with firm roots in functional programming. We developed the iTask system, because of a number of perceived issues with contemporary WFMSs. Their complex nature makes it very hard to correctly create a complete application from the partial application that is generated by them. Furthermore, contemporary WFMSs use special purpose (mostly graphical) specification languages to enable the rapid development of a workflow framework. Unfortunately, these formalisms often offer limited expressiveness. 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. Consequently, workflow models cannot easily use this data to parameterize the flow of work. This results in more or less pre-described workflows that cannot be dynamically adapted. Third, these dedicated languages usually offer a fixed set of *workflow patterns* [1]. 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 specification language probably cannot cope with it either. Fourth, and related, is the fact that functionality that is not directly related to the main purpose of the special purpose language is hard to express. 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, the iTask system is a *domain specific language* that is *embedded* in a textual, formal general purpose *programming language* as a workflow specification language. This allows us to address all computational concerns within the workflow specification and provides us with general recursion. We use a *functional* language, because it offers 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-the-art language that offers fast compiler *and* interpreter technology, generic programming features [2], a hybrid static/dynamic type system [16], which are paramount for generating systems from models in a type-safe way. Workflows modeled in the iTask system result in complete workflow applications that run on the web distributed over server and client side [14]. Clean and the iTask system can be found at `http://clean.cs.ru.nl/` and `http://itask.cs.ru.nl`.

The remainder of this paper is organized as follows. We present the iTask system in Sect. 2 and give a case study in Sect. 3. We discuss our experience in Sect. 4 and 5. Related work is discussed in Sect. 6. We conclude in Sect. 7.

2 Overview of the iTask system

The iTask system is a scientific prototype of a WFMS. It is also a real-world application that deploys and coordinates contemporary web technology. The main reason for using web technoloy is that WFMSs are by nature distributed, multi-user, and heterogeneous software systems. The iTask system is a library made in the functional programming language Clean. The specifications that serve as input to the iTask system are expressed as a domain specific language embedded in Clean. We have adopted the practice in the functional programming community to provide a library offering a set of *combinator functions* and *primitive functions* to allow for compositional, higher-order, parameterized model specifications.

In order to give an impression of the combinators that a workflow engineer can use, Fig. 1 shows a few of the combinator functions and types that constitute the iTask domain specific language (for reasons of presentation, the types have been slightly simplified).

```
:: Task a          // Task is an opaque, parameterized type constructor

// Sequential composition:
(>>=) infixl 1  :: (Task a) (a → Task b)    → Task b   | iTask a & iTask b
return          :: a                         → Task a   | iTask a

// Splitting-joining any number of arbitrary tasks:
anyTask         :: [Task a]                  → Task  a  | iTask a
allTasks        :: [Task a]                  → Task [a] | iTask a

// Task assignment to workers:
class    (@:) infix 3 w :: w (String,Task a) → Task a   | iTask a
instance @: User, String
```

Figure 1: A snapshot of the iTask combinator functions.

A task is an expression of the opaque (hidden), parameterized type `Task a`. Here, `a` is a type parameter that can be instantiated with any conceivable first order type. It represents the type of the value that is produced by the task. Hence, a task (expression) of type `Task a` is a task that, once it has been performed, produces a value of type `a`.

Tasks can be combined *sequentially*. The infix combinator `>>=` and `return` function are the standard *monad* combinators [11]. Task `t >>= f` first performs task `t`, which eventually produces a value of type `a`. This value can be used by the *function argument* `f`, which can compute any new kind of task expression based on that information. The type demands that `f` eventually produces a value of type `b`, which is also the final result of `t >>= f`. The task `return v` only produces value `v` without any effect.

Any number of tasks `ts` $= [$`t`$_1 \ldots$ `t`$_n]$ $(n \geq 0)$ can be performed in parallel

and synchronized (also known as *splitting* and *joining* of workflow expressions): `anyTasks ts` and `allTasks ts` both perform all tasks `ts` simultaneously, but `anyTasks` terminates as soon as *one* task of `ts` terminates and yields its value, whereas `allTasks` waits for completion of *all* tasks and returns their values.

Tasks can be assigned to workers. The expression `w @: (l,t)` assigns task `t` to worker `w`. Here `l` is a descriptive label (like the subject field in an e-mail message). The infix operator `@:` is overloaded in the identification value of the worker, which can be a value of type `User` (a predefined iTask type), or by means of the user name (`String` value).

A more detailed description of these combinators is out of scope of this paper, but in Sect. 3 we give a complete example of a small, yet realistic and complex workflow that uses many of the above combinators. The crucial points are that first, all combinator functions are parameterized and statically type checked with the data that flows along the tasks. Second, tasks can inspect this data and change the control flow accordingly. Third, there is no limit on the type of the data that is passed along, provided that suitable generic functions (see Sect. 5) are available. This is expressed by means of the type class context restrictions (`| iTask` ...). Fourth, several combinators to express iteration are included in the iTask library. However, because the iTask system is a library embedded in Clean, the workflow engineer can define new combinators and even define recursive workflows if desired.

In addition to combinators that combine task expressions in new ways, the workflow engineer also needs primitive iTask functions. Fig. 2 shows some.

```
// Worker interaction:
enterInformation     :: question          → Task a      | html question & iTask a
updateInformation    :: question a        → Task a      | html question & iTask a
showMessage          :: message           → Task Void   | html message
chooseTask           :: question [Task a] → Task a      | html question & iTask a

// Worker administration:
chooseUsersWithRole:: question String    → Task [User] | html question
```

Figure 2: A snapshot of the iTask primitive combinator functions.

The archetypical primitive iTask combinator is `enterInformation q` which, when performed, presents the current worker with a form to create a new value of type `a`. Here, `q` is a guiding prompt for the worker. Fig. 3 gives an example of a form for the type `Person`. `updateInformation q v` is similar, except that the value `v` acts as initial content of the form. The `showMessage` combinator displays a message to the user. With `chooseTask` the user can choose a task to be performed from a list of tasks. In order to dynamically delegate work to users in the system, a workflow needs to have access to the worker administration. With the combinator function `chooseUsersWithRole` the user is given a list of current workers, and she can make a selection.

The overview of the iTask combinators here is just a selection enabling us

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

enterPerson :: Task Person
enterPerson = enterInformation "Enter Information"
```

Figure 3: A standard form editor generated for type `Person`.

to present the example used in Sect. 3. There are many more combinators that we cannot discuss here due to lack of space: combinators for the dynamic creation and control of workflow *processes*, combinators to raise and handle *exceptions* (stop a running workflow, inform all collaborators and start an alternative workflow), and combinators which allow to *change* workflows *at execution time* (replace a workflow on-the-fly by another workflow yielding a result of the same type). These features are necessary to handle realistic workflow cases.

Finally, iTask is embedded in Clean. This provides the workflow engineer with many abstraction techniques that are common practice in functional programming: tasks can be polymorphic, use higher-order functions, can be parameterized, and even higher-order workflows can be created (tasks that have tasks as parameter or result). This yields a high degree of reusability and customization. As a final example, iTask provides a core combinator function, `parallel` that is used in the system to define many other split-join combinators such as `anyTask` and `allTasks` that were shown earlier. Its type signature is:

```
parallel :: ([a] → Bool) ([a] → b) ([a] → b) [Task a] → Task b | iTask a & iTask b
```

`parallel c f g ts` performs all tasks within `ts` simultaneously and collects their results. However, as soon as the predicate `c` holds for any current collection of results, then the evaluation of `parallel` is terminated, and the result is determined by applying `f` to the current list of results. If this never occurs, but all tasks within `ts` have terminated, then `parallel` terminates also, and its result is determined by applying `g` to the list of results.

3 Ordering example

To demonstrate the expressive power of iTask, we present an *ordering* 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`).

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

Determining the required amount of goods proceeds in a number of steps:

```
getAmount :: Task Amount                                                    3.
getAmount                                                                   4.
= chooseTask "Decide how much we need"                                      5.
    ["Decide yourself"   @>> enterInformation "Enter the required amount"   6.
    ,"Let others decide" @>> determineOthers]                               7.

determineOthers :: Task Amount                                              8.
determineOthers                                                             9.
=              chooseUsersWithRole "Select institutes:" "Institute"         10.
  >>= λusers → allTasks [ user @: ("Amount request", getAmount)             11.
                        \\ user ← users                                     12.
                        ]                                                   13.
  >>= λothers → updateInformation "Enter required amount" (sum others)      14.
```

First, with `chooseTask` the user can choose to enter the amount herself or to ask others to determine this amount. `@>>` is used to give a task a (displayable) label. In `determineOthers`, with the task `chooseUsersWithRole` (line 10) a set of users (of type `User`) which fulfil a certain role, in this case institutes, is selected by the user. Each of the selected institutes on their turn may enquire other institutes *recursively* in parallel (using the `allTasks` combinator) how many goods they need (lines 11-13). The recursive call `getAmount` has as effect that each of the chosen institutes can ask other institutes for the same thing, and so on. Given the amount determined by others, an institute may alter the final amount it wants to have (line 14). `Amount` is a non-negative `Int`:

```
:: Amount :== Int                                                           15.
```

Once the amount of goods is established, the workflow can continue by inviting offers from a collection of candidate suppliers:

```
inviteOffers :: Amount → Task [(Supplier,Amount)]                           16.
inviteOffers needed                                                         17.
=              chooseUsersWithRole "Select suppliers:" "Supplier"           18.
  >>= λsups → parallel enough (maximum needed) id                           19.
               [sup @:("Order request", updateInformation prompt needed     20.
                                        >>= λa → return (sup,a))            21.
               \\ sup ← sups                                                22.
               ]                                                            23.
where enough as = sum (map snd as) >= needed                                24.
      prompt    = "Request for delivery, how much can you deliver?"         25.
```

This collection is determined first (line 18). Each supplier can provide an amount (line 20). This is again done in parallel (line 19-23). The termination criterium is the `enough` predicate which is satisfied as soon as the sum of

provided offers exceeds the requested amount (line 24). The canonization function `maximum` is discussed below. Hence, the result of this task is a list of offers. Each offer is a pair of a supplier and the amount of goods that it offers to deliver. A supplier is just a user:

```
:: Supplier :== User                                                    26.
```

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.

```
maximum :: Amount [(Supplier,Amount)] → [(Supplier,Amount)]            27.
maximum needed offers   = [(sup,exact) : rest]                          28.
where                                                                   29.
    [(sup,_) : rest]    = sortBy (λ(_,a1) (_,a2) → a1 > a2) offers      30.
    exact               = needed - sum (map snd rest)                   31.
```

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

```
placeOrders :: [(Supplier,Amount)] → Task [Void]                        32.
placeOrders offers                                                      33.
=   allTasks [sup @: ("Order placement", showMessage ("Please deliver " <+ a))   34.
             \\ (sup,a) ← offers                                        35.
             ]                                                          36.
```

The overloaded infix operator `<+` converts its right-hand argument to a string and glues it to the given left-hand argument. It is part of the iTask system.

In order to complete the case study, the `getSupplies` workflow needs to be passed to the iTask run-time system as a workflow that returns `Void`:

```
Start :: *World → *World                                                37.
Start world  = startEngine [workflow] world                             38.
where                                                                   39.
    workflow = { name     = "Ordering example"                          40.
               , label    = "Collect ordering info and make the order"  41.
               , roles    = []                                          42.
               , mainTask = getSupplies >>= λ_ → return Void            43.
               }                                                        44.
```

4 Experience with the iTask language

iTask is a prototype language. We have investigated its expressiveness by means of constructing examples as well as larger case studies, for instance a conference management system [13]. The next step is to investigate its use in demanding environments that concern crisis management situations, in a project with the Netherlands Defense Academy. In this section we report on our experience in using the iTask specification language.

iTask is built on a single, powerful, concept

In iTask, everything is constructed as (a combination of) a task. The notion of a task and the combinators we use have a clear semantics [7]. A task represents work that needs to be performed, and abstracts over the way the task is composed out of sub-tasks and the order in which these sub-tasks are being evaluated. No matter how complex a task may be, for the programmer a task remains a unit of work returning a value of type (`Task a`) once the task as a whole is terminated. The result of a task can be used as input for other tasks. The coordination of tasks is defined by means of combinators.

A task represents work that needs to be performed. This work can be anything that is required by the workflow case, such as connecting to a legacy information system, calling a web service, or arbitrary foreign code. For instance, for access to information stored in standard information systems, we have developed a systematic conversion between an information model defined in e.g. ORM (Object Role Model) and Clean data type definitions. This enables the automatic conversion between values of these types and the corresponding values stored in a relational database [8], without the need for explicit SQL programming. As another example, for the type `GoogleMap`, the basic task `enterInformation`

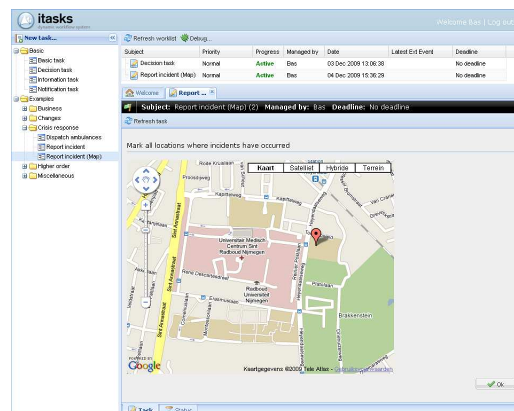

Figure 4: An iTask for manipulating a map

will show a standard Google Map in which the end user can scroll and place markers (Fig. 4). User manipulations of the map are automatically kept track of and are reflected in the `GoogleMap` data structure. No extra effort is needed in the workflow specification other than using the type.

In this way, everything can be considered to be a task. An iTask specification uses combinators to coordinate tasks, and hence one can use the iTask language as a web coordination language as well.

iTask is a declarative language

We want the specification of a workflow to be declarative and hence to abstract from details as much as possible. Given an iTask workflow specification, the iTask system automatically generates all required web forms, handles all user data entry, storage of intermediate results, task distribution to specified workers, and handles all coordination. Also the precise way information is displayed in the browser is not specified in the workflow, but delegated to the client. To further enable abstraction over lay-out, we offer several primitives in the iTask library for basic interaction steps. For instance, in addition to `enterInformation`, there are basic primitives like `enterChoice` and `enterMultipleChoice`. The advantage of having different primitives for such basic interaction steps is that the workflow specification becomes more readable while the representation and lay-out can again be delegated to the client. Due to abstraction, the workflow engineer can concentrate on specifying the workflow. This promotes rapid prototyping of workflow applications.

iTask is more than Clean

iTask is an embedded domain specific language and inherits all language aspects of its host Clean. In particular, these are the strong type system, higher-order functions, lazy and strict evaluation, and the module system. All computational and algorithmic concerns can be dealt with in the Clean language. iTask is also more than Clean because workflows are inherently sequential, distributed, multi-user, concurrent systems and the Clean standard supports neither of those. Also, to model realistic workflow cases, one needs to address exceptions and dynamic change. Again, these concepts are absent in native Clean (see also Sec. 5). Each of the required concepts of the embedded language are challenging to add to native Clean. Nevertheless, this experiment shows that it is possible to embed a workflow language in a host that offers entirely different concepts.

iTask has higher-order tasks

A task in Clean of type `Task a | iTask a` effectively works for all first order types `a`. In particular, it works for the type `Task` itself, which means that tasks can be higher order: the result of a task might be a task which can be dynamically and interactively constructed. In this way meta programming (doing tasks that have as goal to define new tasks) can be accomplished. A task thus created can be given as argument to other tasks which can decide to evaluate it or to use it in the construction of an even more complex task. It is very unlikely that an ad-hoc domain specific workflow language has the ability to deal with advanced notions such as higher functions and tasks, and this feature is therefore missing in all commercial workflow systems. Embedding a workflow language in a language like Clean really pays off here.

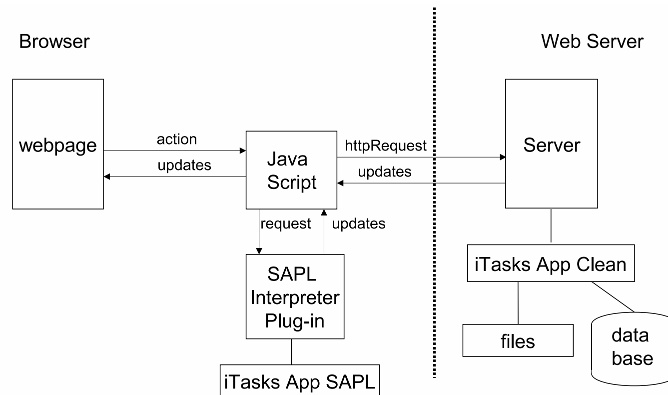

Figure 5: The architecture of an iTask application

5 Experience with Clean as host language

In this section we focus on our experience with using Clean as host language and implementation vehicle to embed iTask. An iTask specification results in a web application. The architecture of this web application is given in Fig. 5.

Smart combinators

iTask is a workflow language and is hence inherently sequential, distributed, multi-user, and concurrent. It needs to handle exceptional situations and dynamically changing workflows. The host language Clean offers no native support for these concepts. When developing such a language in the traditional way, one would develop a grammar, semantic rules, perhaps a type system, a compiler and/or interpreter, code generator, and so on. This is a huge amount of work. In this project we have taken a different route: when designing a language, one needs to define the semantic rules. Semantic rules can be represented in a natural way by means of functions. If one takes care in designing these rules in a compositional way, then these form a set of *smart combinator* functions. In this way one can obtain a compositional language implementation almost for free. This decreases the implementation effort of a new language significantly.

The combinators have several obligations in the iTask system. First, the combinators yield the current status (and hence GUI) at any moment during execution. For example, the iTask system can evaluate the expression `t >>= f` even if task `t` is not finished yet. The iTask system does this by creating a default value of the proper type for the whole expression `t >>= f`. In this way the status of all tasks defined in a workflow can be inspected, but only the values of the finished tasks are taken into account. Second, a new workflow is calculated by the combinators given the finished tasks. Third, each combinator stores its current state in memory and uses it for handling the next event from the participating workers.

Smart tasks

The iTask language is a declarative language. This implies that we want to generate as much boilerplate code as can be possibly done from an iTask specification. In iTask this has been realized by using the generic programming features of Clean [2]. Tasks require the availability of a collection of generic (kind indexed, type driven) functions. These generic functions are used to generate all kinds of functionality automatically, such as the generation of web forms, the handling of user updates of such forms, the storage and retrieval of information, the serialization and de-serialization of data and functions. The generic functions are predefined in the iTask library. To use them for a certain type, however, one needs instances for that type for all the generic functions being used. As a result a task can be applied to values of *any* type, as long as instances for this type have been defined for all generic functions the task is depending on. The Clean compiler is able to generate instances for these generic functions for (almost) any (non opaque) type fully automatically. Clean is special in this respect. In Haskell e.g. generic functions can be constructed using special pre-processors like template Haskell.

It should be noted that a great deal of the facilities for which we have used generics in our project can be done in a programming language that offers introspection and code generation facilities. One significant advantage of using generics is its firm integration with the static type system of Clean.

Smart serialization

An iTask application is a web application that runs on the server side. This application must handle every possible user request from any possible web browser that connects with the application. After an event is handled, the web application terminates and is started all over again by the web server when new user events arrive. Hence, an iTask application needs to fully recover its previous state to compute the proper response. Conceptually, this amounts to reconstructing the *task tree* that reflects the current state of computation of the workflow. The *nodes* of a task tree are formed by the combinators in the task that is being computed, and the *leaves* of a task tree are the primitive tasks. Evaluation of a workflow amounts to *rewriting* this task tree as dictated by the combinators. The task tree can become very big. Hence, a naive implementation of task tree rewriting for iTask applications is not realistic. Instead, we have incorporated a number of optimizations that are required to obtain an efficient and scalable implementation. We briefly discuss two of the most important optimizations.

The first optimization is based on the observation that most rewrites affect only a local part of the task tree. Hence, for these rewrites it is not necessary to reconstruct the entire task tree, but only the sub task tree that can be affected. Because an iTask application terminates after handling an event, we need to be able to store and read any sub tree that is currently being rewritten. Tasks and combinators are implemented as state transition functions, hence we need to be able to store functions. Clean offers a hybrid type system, and statically typed

expressions can be turned into a dynamically typed expression (of static type `Dynamic`) and the other way around. Dynamics can be stored to disk and it is even possible to read in a dynamic stored by some other Clean application.

The second optimization is based on the observation that many computations do not *have to be done* at the server side, but can also be done on the web client side. Hence, clients need to be able to run tasks, which amounts to running Clean code. To implement this, the Clean compiler generates *two* executable instances from a single source. The first instance is a Clean executable that runs on the server, and the second instance is a SAPL program to be executed by the SAPL interpreter [6] that is running as a Java applet at the client side. At run-time it can be decided where to execute what. Any function or task can be shifted from server to client. For this purpose we again use dynamics in Clean to serialize functions and expressions as SAPL programs at the server side and interpret them at the client side. For details we refer to [14].

6 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. *Objects* accumulate the progress made in a 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 *done* when the workflow procedure is applied, and what further workflow procedures should be *processed* 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; reasoning about the evaluation of an iTask program is reasoning about the combinators instead of the collection of clauses.

Brambilla *et al*[4] 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 [4] and can only be done ad-hoc.

Pešić and van der Aalst [10] 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 [9]. 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*[3] 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 mod-

els). 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 [4] and [10]: 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*[15] 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 to connect them.

7 Conclusions

In this paper we report on our experience in using the lazy, pure, functional language Clean as embedding language to specify and create web-based workflow iTask applications. Although the iTask combinator language is embedded as a library in Clean, it is by no means a *shallow* embedding, i.e. the meaning of the embedded language is not a straightforward extension of the host language. The result *is* a new language for defining workflow applications. This new language provides the workflow engineer with concepts to seamlessly merge data flow with control flow (exemplified by the >>= combinator), use higher-order tasks (tasks that can create, manipulate, and pass around tasks), in a compositional way. The evaluation order of the workflow is controlled by the iTask combinators and dictated by the needs of the workflow engineer (by using sequential and generalized parallel split-join patterns as well as recursion). It is important to observe that this evaluation order is very different from the lazy evaluation order of the host language *and* that one can add new combinators within iTask to capture other evaluation orders when needed. The iTask system is very general and serves as a coordination language to control and unify all tools that are used to realize the system. Specifications inherit the terseness of their host language.

We have used many state-of-the-art programming language techniques to obtain this result: *generic programming* to handle boilerplate code generation (including foreign code) in a type-directed way, *dynamic types* to handle arbitrary (higher-order) data structures which origin need not be the source program

itself, and *higher-order functions* which permeate through the entire design, implementation, and resulting language. The entire system is statically typed. Although the boilerplate code generation aspects can be realized in other programming languages that support some form of inspection, we have shown in this project that the task of embedding a language (however alien) is one that fits functional programming languages like a glove.

References

[1] Wil van der Aalst, Arthur ter Hofstede, Bartek Kiepuszewski, and Ana Barros. Workflow patterns. QUT technical report, FIT-TR-2002-02, Queensland University of Technology, Brisbane, Australia, 2002.

[2] 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.

[3] Birger Andersson, Maria Bergholtz, and Ananda Edirisuriya. A Declarative Foundation of Process Models. In Oscar Pastor and João Falcão e Cunha, editors, *Proceedings 17 Int'l Conference on Advanced Information Systems Engineering, CAiSE 2005*, volume 3520 of *LNCS*, pages 233–247. Springer-Verlag, 2005.

[4] Marco Brambilla, Jordi Cabot, and Sara Cornai. Automatic Generation of Workflow-Extended Domain Models. In Gregor Engels, Bill Opdyke, Douglas C. Schmidt, and Frank Weil, editors, *Proceedings Model Driven Engineering Languages and Systems, 10th Int'l Symposium, MoDELS 2007*, volume 4735 of *LNCS*, pages 375–389. Springer-Verlag, 2007.

[5] Zef Hemel, Ruben Verhaaf, and Eelco Visser. WebWorkFlow: an object-oriented workflow modeling language for web applications. In K. Czarnecki, I. Ober, J.-M. 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] Jan Martin Jansen, Pieter Koopman, and Rinus Plasmeijer. Efficient interpretation by transforming data types and patterns to functions. In Henrik 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] 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.

[8] Bas Lijnse and Rinus Plasmeijer. Between types and tables - Using generic programming for automated mapping between data types and relational databases. 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.

[9] Maja Pešić. *Constraint-based workflow management systems: shifting control to users*. PhD thesis, Technical University Eindhoven, 8, October 2008.

[10] Maja Pešić and Wil van der Aalst. A declarative approach for flexible business processes management. In Johann Eder and Schahram 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.

[11] Simon Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th International Symposium on Principles of Programming Languages, POPL'93*, pages 71–84, Charleston, SC, USA, January 1993. ACM Press.

[12] 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.

[13] 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*, volume 5832 of *LNCS*, Center Parcs "Het Heijderbos", The Netherlands, 19-24, May 2008. Springer-Verlag.

[14] 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.

[15] Irene Vanderfeesten, Hajo Reijers, and Wil 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.

[16] Martijn Vervoort and Rinus Plasmeijer. Lazy dynamic input/output in the lazy functional language Clean. In Ricardo Peña and Thomas Arts, editors, *Selected Papers of the 14th International Symposium on the Implementation of Functional Languages, IFL'02*, volume 2670 of *LNCS*, pages 101–117. Springer-Verlag, September 2003.