Getting a Grip on Tasks that Coordinate Tasks

Rinus Plasmeijer
Radboud University Nijmegen
Nijmegen, the Netherlands
rinus@cs.ru.nl

Bas Lijnse
Radboud University Nijmegen
Nijmegen, the Netherlands
b.lijnse@cs.ru.nl

Peter Achten
Radboud University Nijmegen
Nijmegen, the Netherlands
P.Achten@cs.ru.nl

Steffen Michels
Radboud University Nijmegen
Nijmegen, the Netherlands
s.michels@science.ru.nl

ABSTRACT

Workflow management systems (WFMS) are software systems that coordinate the tasks human workers and computers have to perform to achieve a certain goal. The tasks to do and their interdependencies are described in a Workflow Description Language (WDL). Work can be organized in many, many ways and in the literature already more than hundred of useful workflow patterns for WDL's have been identified. The iTask system is not a WFMS, but a combinator library for the functional language Clean to support the *construction* of WFMS applications. Workflows can be described in a compositional style, using pure functions and combinators as self-contained building blocks. Thanks to the expressive power of the underlying functional language, complex workflows can be defined on top of just a handful of core task combinators. However, it is not sufficient to define the tasks that need to be done. We also need to express the way these tasks are being supervised, managed and visualized. In this paper we report on our current research effort to extend the iTask system such that the coordination of work can be defined as special tasks in the system as well. We take the opportunity to redesign editors which share information and the combinators for defining GUI interfaces for tasks, such as buttons, menu's and windows. Even though the expressiveness of the resulting system increases significantly, we are certain that the number of core combinators can be reduced. In this paper we argue that only two general Swiss-Army-Knife higher order functions are needed to obtain the desired functionality. This simplifies the implementation significantly and increases the maintainability of the system. In this paper we discuss the design space and decisions that lead to these two general functions for constructing tasks.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features

General Terms

Languages

Keywords

embedded domain specific language, functional combinator library, workflow system

1. INTRODUCTION

Workflow management systems (WFMS) are software systems that coordinate, generate, and monitor tasks performed by human workers and computers. A concrete workflow ensures that essential actions are performed in the right order. The purpose of the iTask system [5] is to support the construction of WFMS applications. It distinguishes itself from traditional WFMSs. First, the iTask system is actually a monadic combinator library in the pure and lazy functional programming language Clean. The constructed WFMS application is embedded in Clean where the combinators are used to define how tasks can be composed. Tasks are defined by higher-order functions which are pure and self contained. Second, most WFMSs take a workflow description specified in a workflow description language (WDL) and generate a partial workflow application that still requires substantial coding effort. An iTask specification on the other hand denotes a full-fledged, web-based, multi-user workflow application. It strongly supports the view that a WDL should be considered as a complete specification language rather than a partial description language. Third, despite the fact that an iTask specification denotes a complete workflow application, the workflow engineer is not confronted with boilerplate programming (data storage and retrieval, GUI rendering, form interaction, and so on) because this is all dealt with using generic programming techniques. Fourth, the structure of an iTask workflow evolves dynamically, depending on user-input and results of subtasks. Fifth, in addition to the host language features, the iTask system adds higher-order tasks (workflow units that create and accept other workflow units) and recursion to the modelling repertoire of workflow engineers. Sixth, in contrast with the large catalogue of common workflow patterns [1], iTask workflows are captured by means of a small number of core combinator functions.

In this paper we reflect on these core combinators and the functionality they offer. Although complex workflows can be defined in a declarative style, one would like to have more flexibility in controlling the tasks one is working on. For instance, when a task is delegated, someone might want to monitor its progress. In the current system the delegator gets this information and she also obtains the power to change the properties of the delegated task, such as its priority, or, she can move the task to the desk of someone else. This is often useful, but is not always what is wanted. Perhaps one would like to inform other people involved as well. One also would like to define what kind of information is shown to a particular person and define what a manager can do with the tasks she is viewing. Controlling tasks can be seen as tasks as well and one would like to have combinators to programme their behaviour. In particular one would like to define control interfaces that show what goes on and which can be used to manage the tasks involved. The extended iTask system described in [4] appears to be a good starting point for defining such interfaces. In that paper we extended the system with new combinators given the ability to define GUIs for tasks. Furthermore we showed that it is possible to share information between tasks. Tasks can communicate with each other by modifying shared information.

Adding all these extensions to the iTask system can easily lead to a system with an excessive number of core combinators. This leads to high maintenance costs and hampers formal reasoning. Fortunately, the desired functionality can be obtained with only a very few powerful combinators with which the simplicity of the system can be retained and the maintainability can be improved. It is the thesis of this paper that we can do with only two new general purpose elements.

The remainder of this paper is organized as follows. First, we describe the current core iTask system (Section 2) and explain its usage and shortcomings by means of small, yet illustrative examples (Section 3). Based on this analysis we identify the requirements that should be satisfied by the new iTask system and argue that they can be realized with only two new general constructs with which all current constructs can be defined (Section 4). In the conclusions we briefly discuss the current situation (Section 5).

2. THE ITASK CORE SYSTEM

In this section we give a brief overview of the iTask system. The kernel of the iTask system consists of basic tasks and combinator functions for combining tasks. On top of them, an API is defined. This API offers some notational convenience and allows to define workflows in a more verbose style to enhance the communication with domain experts, which are not likely to be functional programmers. In this paper we abstract from this API and focus on the iTask kernel.

2.1 Basic Tasks

Basic tasks are units of work of the opaque, parameterized type `Task a`: the type parameter `a` is the type of the result value that is committed to the workflow when the task has finished. In principle, any unit of work in daily life can be modeled as a basic task. It can be a call to a service on a web server, a system call, or a form to be filled in by a worker in a browser. The latter is the most interesting one in this context, since human interaction is a key feature of any workflow system. In this paper we restrict ourselves to this basic task only, for which we provide the function `edit`.

```
edit :: String a → Task a | iTask a
```

Note that in Clean the arity of functions is shown explicitly by separating argument types by spaces instead of →. An editor is created with `edit prompt va`. The `prompt` argument provides the worker with information about the purpose of this task. When applied to an initial `va` of some type `a`, it creates a GUI in which the worker can inspect and alter the given value arbitrarily many times. An editor can create and handle such a GUI for any first-order type `a`. It uses a set of type indexed generic functions (hence the context restriction `| iTask a`) which are derived by the compiler automatically. The iTask system guarantees that only values of type `a` can be created. This continues until the worker decides to commit the value to the workflow, which terminates the task `edit prompt va`.

2.1.1 Example

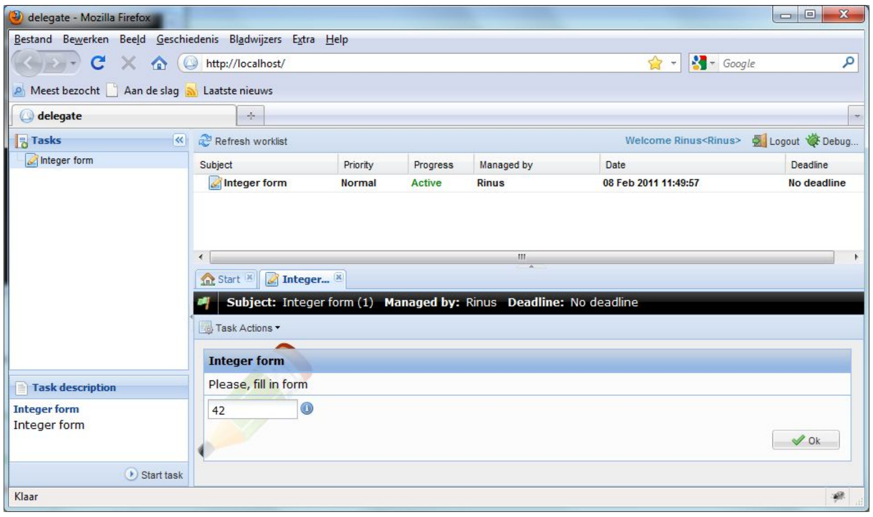
What follows is a very small iTask example: a task `enterInt` using the `edit` function to create a form for filling in an `Integer` number. The generic function `initialValue` yields an initial value for any first order type.

```
module example

import iTask

Start world
  = startEngine [workflow "Integer form" enterInt] world

enterInt :: Task Int
enterInt = edit "Please, fill in form" initialValue
```

To turn this task description into an executable WFMS one has to import the `iTask` library. The main function in Clean is called `Start` which obtains a (unique) world as argument which is used for the pure communication with the impure outside world. The library function `startEngine` takes a list of workflow specifications and creates a web-based workflow system for it (see Figure 1). Any task can be promoted to become a workflow with the function `workflow`. Such a workflow is added to the list of workflows in the left workflow start-pane. A workflow can be started by the worker arbitrary many times just by clicking on its icon. The tasks a worker has to do appear in the task-list pane, similar to a list of incoming emails. By clicking on an item in the list, a task-pane is opened allowing a worker to work on her tasks in arbitrary order.

The `edit` function can be used on any first-order type. In the example below we show the form it creates for a list of `Persons`, where `Person` is some user-defined record type. One only has to change the type of the application of `edit`. Furthermore one has to ask the compiler to **derive** the generic functions for the types involved. The resulting form is shown in Figure 2.

```
module example2

import iTask

:: Person   = { firstName   :: String
              , surName     :: String
              , dateOfBirth :: Date
              , gender      :: Gender
```



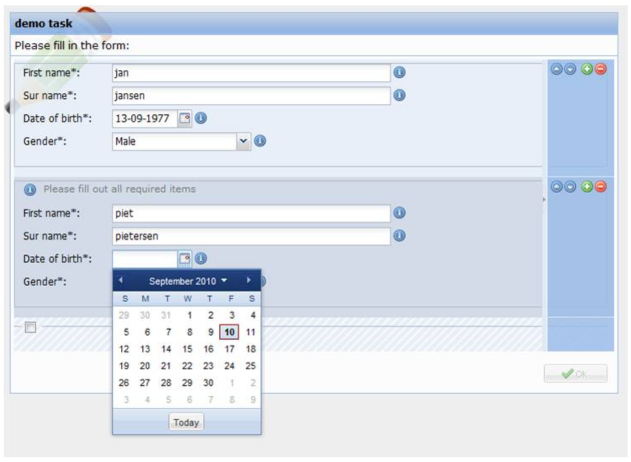
Subject	Priority	Progress	Managed by	Date	Deadline
Integer form	Normal	Active	Rinus	08 Feb 2011 11:49:57	No deadline

Subject: Integer form (1) **Managed by:** Rinus **Deadline:** No deadline

Integer form
Please, fill in form
42

Integer form
Integer form

Figure 1: A screen-shot of the iTask browser interface



demo task
Please fill in the form:

First name*: jan
Sur name*: jansen
Date of birth*: 13-09-1977
Gender*: Male

Please fill out all required items

First name*: piet
Sur name*: pietersen
Date of birth*:
Gender*:

Figure 2: The form generated for editing a list of `Persons`

```
                  }
:: Gender     = Male | Female

derive class iTask Person, Gender

Start world
  = startEngine [workflow "Person form" enterPersons] world

enterPersons :: Task [Person]
enterPersons = edit "Please, fill in the form" initialValue
```

2.2 Core Combinators

In this paper we focus on the iTask core combinators. The semantics is formally defined in [6].

```
// assigning properties to a task:
(@:)    infixr 5 :: p (Task a) → Task a | property p & iTask a

// sequencing of tasks with a monad:
(>>=)   infixl 1 :: (Task a) (a → Task b) → Task b | iTask b
return           :: a                      → Task a | iTask a

// defining parallel tasks: parallel-or and parallel-and:
(-||-) infixr 3 :: (Task a) (Task a) → Task a      | iTask a
(-&&-) infixr 4 :: (Task a) (Task b) → Task (a, b) | iTask a
                                                   & iTask b
```

Properties can be assigned to a task by using the `@:` combinator. In general, the properties that can be set are captured by the type class `property`. Examples of predefined properties are: an identification of a worker or group of workers to which the task is assigned, the priority of a task, and it's deadline. One can assign user-defined properties (type `String`) as well. The properties are inherited by all subtasks of a task, unless other properties are assigned to them via a `@:` operator. We call `@:`-annotated tasks *main tasks*.

To compose tasks sequentially, the monadic combinators `return` and `>>=` [7] are provided. The task `return va` succeeds immediately and commits its value `va`. In `ta >>=` λ`va` $\rightarrow$ `tb`, the task `ta` is evaluated first. When this task commits its value, say `va`, it is passed to `tb` to compute the next task to be executed.

To compose tasks in parallel, the combinators `-||-` and `-&&-` are provided. A task constructed using `-||-` is finished as soon as either one of its subtasks is finished, returning the result of that task. The combinator `-&&-` is finished as soon as both subtasks are finished, and pairs their results.

The iTask system actually offers a couple of additional core combinators: namely for managing workflow processes and for exception handling. However, their functionality is not important for this paper.

In the next section we give some examples of iTask workflow specification using the core combinators described above, and use these examples to explain how the combinators can be further improved.

3. THE EXPRESSIVE POWER AND LIMITATIONS OF THE COMBINATORS

With the iTask core combinators, complex workflows can be defined which cannot be expressed in traditional WFMS. However, there is still room for improvement. We present some examples, explain what their effect is, and discuss what is missing.

3.1 Coordination of Tasks

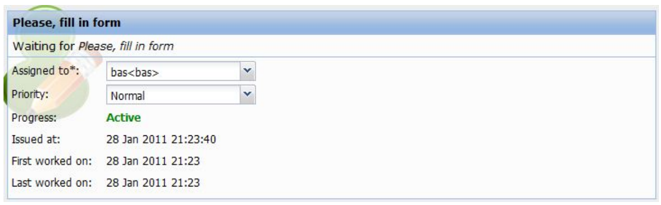
Our first example is a higher order task: we define a task which delegates a given task to someone else. Notice that in most classical WFMSs, such higher order tasks cannot be defined.

```
delegate :: (Task a) → Task a | iTask a
delegate task
=                    selectUsers
    >>= λworker →  worker @: task
    >>= λresult →  edit "Check result:" result

Start world
= startEngine [workflow "Delegate" (delegate enterPersons)]
              world
```

In the `delegate` task there are initially two people involved: a *delegator* delegating work, and a *worker*, to whom the work is delegated. There are three sub-task steps in this delegate task, sequentialized by the bind combinator (`>>=`). When the task `delegate` is performed by the delegator, she first has to select a worker. The library function `selectUsers` displays a list of administrated users the delegator can select from. In the next step the task to delegate is assigned (`@:`) to the chosen worker. When the worker has finished the task, its result is shown to the delegator again. She can correct the returned result with the `editor`, and when she is happy and finishes, the whole delegation task is completed as well. The `delegate` workflow is a clear and concise specification which can be applied to *any* task. In this particular example, `enterPersons` is the task being delegated to the worker.

Yet there is something missing in this specification as well. It is clear what the worker has to do (see Figure 2), but what do we show to the delegator in the meantime? When the work is being delegated, the delegator might be interested in how the work is proceeding. Perhaps she also wants to do something with the information she sees.

In the first version of the iTask system, we just informed the delegator in the task-pane that her delegation task is waiting for the worker to finish. In the current version, we have used the capability to change tasks under execution [6] to turn this passive role of the delegator into an active one. The delegator is offered a predefined control screen as displayed in Figure 3.



Please, fill in form
Waiting for *Please, fill in form*

Assigned to*:	bas<bas>
Priority:	Normal
Progress:	Active
Issued at:	28 Jan 2011 21:23:40
First worked on:	28 Jan 2011 21:23
Last worked on:	28 Jan 2011 21:23

Figure 3: Default task pane of waiting task

It gives the delegator information about the properties of the delegated task, such as its priority, who is working on it and the last time the task has been worked on. Moreover it gives the delegator the capability to *coordinate* the delegated task: she can change its priority and she can replace the worker by another one, including herself. When a change is made, the worker sees this change immediately when an event is committed to the server. When the task is transformed to some other worker, he can continue with the work of the previous worker, since all the work done so far is retained.

In this example we observe that there are actually *two* tasks involved when a main task is created. The task being assigned is explicitly defined by the workflow engineer, but the coordinating task is predefined and fixed in the iTask library. Since the iTask system is intended as a system for constructing WFMS's, it would be better if the coordinating meta-task is not fixed by the underlying system but can be defined as a task by the workflow engineer as well. This implies that such a task must continuously be provided with the actual status information of the main tasks involved.

With this status information one can display a view to whom it concerns. For instance, in the current system, the delegator is by default turned into a manager, but perhaps this is not desirable at all. It might also possible that several people are interested in the progress of the delegated task. Consider the following example:

```
Start world
= startEngine [workflow "Delegate 2"
                   (delegate (delegate enterPersons))]
              world
```

Here we delegate the delegation-task thus introducing potentially two workers who might be interested in the progress of the actual work.

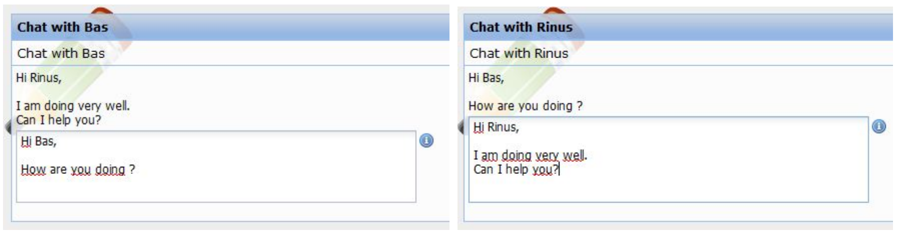
Hence we need to provide the programmer with a combinator with which she can define arbitrary coordination tasks which have a view on the ongoing work and enable to change their properties as wanted.

3.2 Sharing of Information

In the previous section we have identified the need to allow coordination tasks to continuously monitor and alter task (properties). This need extends in a natural way to arbitrary tasks and arbitrary information. Michels *et al.* [4] show that some tasks do need to constantly exchange information while one is working on it. This extends the data flow that is dictated by the current workflow combinators in which information is only propagated to tasks once the information-producing task has been terminated and thus committed its value to the workflow. An appealing example is `chat`.

```
:: Note = Note String

:: View m v = { viewFrom :: m → v, viewTo :: v → m → m }

chat :: User User → Task Void
chat user1 user2
=             createDB (Note "", Note "")
  >>= λref →  (user1 @: editShare ("Chat with "+++user2) a ref)
              -&&-
              (user2 @: editShare ("Chat with "+++user1) b ref)
  >>= λ_ →    return Void
where
  a = { viewFrom = λ(note1, note2) → (Display note2,note1)
      , viewTo   = λ(_,note1) (_, note2) → (note1, note2)
      }
  b = { viewFrom = λ(note1, note2) → (Display note1,note2)
      , viewTo   = λ(_,note2) (note1, _) → (note1, note2)
      }
```

In this example, two workers chat with each other continuously. To make this possible, the text produced by both (of type (`Note`, `Note`)) is stored in a database. It serves as a shared model, and both workers may change the model at the same time. Each worker, however, has its own view on the shared model. For this purpose one has to describe


Chat with Bas

Chat with Bas

Hi Rinus,

I am doing very well.
Can I help you?

Hi Bas,

How are you doing ?

Chat with Rinus

Chat with Rinus

Hi Bas,

How are you doing ?

Hi Rinus,

I am doing very well.
Can I help you?

Figure 4: Workers Rinus and Bas chatting with each other

the mapping between model and view (`viewFrom`) and backwards (`viewTo`) also known as a lens [2]. This implements the well known model-view-controller paradigm [3]. In the `chat` example, the predefined type `Display` is used to prevent one worker to change the information typed in by the other. For a screen shot see Figure 4. Hence, an editing conflict caused when more than one worker changes the same information at the same time cannot arrise in this particular example due to the well chosen view. In general such editing conflicts are possible. The system can prevent the database for becoming inconsistent by producing an error message when somebody is trying to change data which is not up-to-date.

To make this all possible, we need a new kind of editor, like `editShare`, which reads in the current model stored in the database, and converts this information to a view to show in the client. Every time a worker makes a change, the view is converted back to the model and stored in the database. The view others have on this information has to be updated accordingly.

The `editShare` editor has the following type:

```
editShare :: String (View m v) (Ref m) → Task (Maybe m)
                                        | iTask m & iTask v
```

It requires a `String` for prompting, a view of type `v` on a model of type `m` and a reference to the database where the shared model of type `m` is stored. Only if the `editShare` editor finished in a valid state, the final value `mv` is returned as `Just mv`, otherwise `Nothing` is returned.

The ability to share information and provide a specific view on this information is required to define tasks that coordinate other tasks. In this case the state information of the tasks to coordinate has to be provided to the coordinating tasks.

3.3 Adding GUI Elements

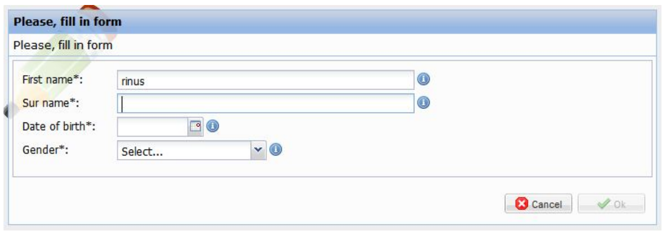
With the standard core editor a form can be generated for any first order type. One can change the values in the form as often as desired. The system ensures that the form can only be filled with values of the demanded type. The standard "OK" button can therefore only be pressed when all required parts of the form have been filled in properly.

Clearly one would like to have the possibility to attach an arbitrary number of buttons to an editor instead of just one. Michels *et al* [4] extend the iTask system and enrich editors with GUI elements such as buttons and menus. Furthermore, several editors can be active at the same time each running in their own window. An example using such an enriched editor, `editA` (for *edit action*), is (see also Figure 5):

```
editA :: String [(Action, (Maybe a) → Bool)] a
      → Task (Action, Maybe a) | iTask a

enterPerson :: Task (Maybe Person)
```

```
enterPerson
      editA "Please, fill in form" myActions initialValue
  >>= λ(event,mbr) → return mbr
where
  myActions        = [(ActionOk,ifValid), (ActionCancel,always)]

  ifValid (Just _) = True
  ifValid _        = False

  always  _        = True
```

The idea is to attach a list of action-predicate pairs to an editor, as shown in the type of `editA`. The predicate defines when the corresponding action can be chosen. An action such as "OK" can only be chosen if a complete form has been filled in, but one can also specify that the entered value has to satisfy additional requirements. Other actions, such as


Please, fill in form

Please, fill in form

First name*: rinus

Sur name*:

Date of birth*:

Gender*: Select...

Figure 5: Editor for type Person with Ok and Cancel buttons.

`Cancel`, should always be possible, regardless what has been entered. It can therefore no longer be guaranteed that such an editor will always return a proper value. Hence, `editA` returns the chosen action and `Maybe` a value. In `enterPerson` two actions are attached to the editor: `ActionOK` which can only be chosen if the form has been filled in completely (guarded with `ifValid`), and `ActionCancel` which can always be chosen (guarded with `always`).

It is better to structure actions using menus when there are many of them. This can be done via task annotations using the operator `<<@`. Here one can define a mapping between actions and menu items. For the actions not mentioned in this mapping buttons are generated. Here we adapt `enterPerson` to use a menu.

```
enterPerson :: Task (Maybe Person)
enterPerson
=     editA "Please, fill in form" myActions initialValue
      <<@ myMenu
  >>= λ(event,mbr) → return mbr
where
    myActions    = [(ActionOk,ifValid), (ActionCancel,always)]
    myMenu       = [Menu "Edit"
                      [MenuItem ActionCancel (Just cancelHotkey)
                   ] ]
    cancelHotkey = {key=C, ctrl=True, alt=False, shift=False}
```

In [4] Michels *et al* propose a special combinator for creating multiple editors in parallel each running in their own window. As we will see in Section 4, we conjecture that all the different types of editors presented so far can be combined into one. Similarly, we can also combine the different ways of creating parallel tasks in one combinator.

3.4 Swiss-Army-Knife Parallel Combinator

The need for more functionality does not necessarily imply that more combinators are required. By using higher order functions, Swiss-Army-Knife combinators can be defined, that strongly reduce the number of needed core combinators. In the current iTask system, the `parallel` combinator is one such example:

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

For instance, the core combinators `-||-` and `-&&-` (Section 2.2) can be replaced by suitable parametrization of `parallel`. The function `parallel predOK someDone allDone taskList` takes a list of tasks (`taskList`) to be executed in parallel, a predicate (`predOK`), and two conversion functions (`someDone` and `allDone`). Whenever a member of `taskList` is finished, its result is collected in a list `results` of type `[a]`, maintaining the order of tasks. Now `predOK results` is computed to determine whether `parallel` should complete, in which case the result is computed by `someDone results`. When all parallel tasks have run to completion, and `predOK` is still not satisfied, then `parallel` also completes, but now with result `allDone results`. We can define `-||-` and `-&&-` as follows:

```
(-||-) infixr 3 :: (Task a) (Task a) → Task a | iTask a
(-||-) ta1 ta2 = parallel (not o isEmpty) first undef [ta1, ta2]
where
    first [a] = a

(-&&-) infixr 4 :: (Task a) (Task b) → Task (a, b)
                | iTask a & iTask b
(-&&-) ta tb = parallel (const False) undef all
                        [ta >>= Left, tb >>= Right]
where
    all [Left a,Right b] = (a,b)
```

Although a Swiss-Army-Knife combinator such as `parallel` can be used to define many different kinds of parallel behaviours, there is room for improvement here as well. With `predOK` one can freely define when the parallel tasks can be stopped, but perhaps one also needs to be able to start new tasks dynamically, because more work is required.

Also there seem to be different categories of work to be done in parallel. One category is formed by tasks different people work on in parallel, such as in the `chat` example. But one can also think of parallel tasks one person works on, each task running in its own window being part of one and the same GUI-application.

4. REDESIGNING THE ITASK CORE SYSTEM

In the previous section we have identified some shortcomings of the current iTask core system as presented in Section 2. One would like to have a more general basic task editor that can be used for ordinary tasks as well as for the coordination of tasks. Furthermore, one would like to have a more general applicable combinator for defining parallel tasks. In this section we argue that it is possible to identify two such general purpose new elements.

4.1 Basic Editor Task Revisited

The basic `edit` task, as defined in Section 2.1, lacks some functionality needed to define coordination tasks. In Sections 3.2 and 3.3, we gave examples of extended basic tasks with additional functionality.

As for task combinators the different variants of editor tasks can be seen as special cases of one Swiss-Army-Knife editor task. For instance, a task not using actions is actually a special case only using the `Ok` action which can only be chosen if the editor is in a valid state. Working on local data can be seen as a special case of working on a shared database, which is only used by a single task and deleted afterwards. A last example is that tasks not using a custom defined view, actually use the identity view (defined as `{viewFrom = id, viewTo = const}`).

Summarized a Swiss-Army-Knife editor task has to meet the following criteria:

1. Edited data can be shared by an arbitrary number of editor tasks, which are possibly carried out by different workers. The system ensures that the data is kept in a consistent state by detecting and reporting edit conflicts.

2. It is possible to edit only a part of the data given to the task. Also the representation shown to the worker might be different than the original data model. This can be achieved by using functionally defined views.

3. An arbitrary number of actions can be attached to each editor task. They are triggered either by buttons or menus which structure is given by annotating the task. A predicate is used to define when an action can be triggered. The task only returns a value if the editor stopped in a valid state.

We are currently implementing all tasks for user interaction in the iTask library (which also includes special tasks not discussed in this paper, for example for making choices), using the same underlying general editor implementation. In the actual implementation some optimizations might be considered. For example, not storing a separate database if it is only used by a single task.

4.2 Core Combinators Revisited

In Section 3.4, we have shown how one single combinator, `parallel`, can be used to create the derived combinators `-||-` and `-&&-`. We have argued in Section 3.1 that delegating work is also a form of parallel task creation. The current shortcoming of delegation is that the iTask system has predefined behaviour to control and coordinate these tasks. The workflow engineer should be able to specify the means of control as (arbitrarily many) additional tasks that coordinate these tasks. We hypothesize that these forms of parallel behaviour can be captured with a single, more general combinator. The combinator needs to meet the following criteria:

1. The number of tasks in the current `parallel` combinator remains constant, and `parallel` can only enforce early termination, not the extension of new tasks. The number of tasks in a parallel setting should not be fixed once and for all, but should adapt to the needs of the current situation.

2. The tasks within the current `parallel` combinator simply perform their duty and as such do not interfere with each other (except ofcourse when using shared communication). Next to these *regular* tasks we introduce *control* tasks. These are also tasks, but, being control tasks, they 'edit' the collection of parallel tasks. In this way, we can replace the predefined behaviour of task delegation and instead leave it to the workflow engineer whether or not to use a predefined control delegation-task or introduce a (number of) custom control task(s).

3. Because the number of both regular and control tasks varies during the evaluation of a parallel group, we need to *share* information about the state of the parallel group. Access to this state is restricted to control tasks only, which is easily achieved using the strong type system.

4. In the current `parallel` combinator, control is limited to *either* early completion (computed by `predOK`) in which case the final task result was computed by `someDone` *or* full completion in which case the final result was computed by `allDone`. In the more general case, we need to decide how to continue whenever a regular or control task runs to completion. Again, this should not be computed by the regular tasks. Instead, we need a function that knows which task has completed, and hence has a result value that needs to be accumulated in the shared state. In addition, this function can decide what should happen with the group of parallel (control and regular) tasks: tasks can be suspended and resumed, they can be removed, replaced, and new (control and regular) tasks can be added to the group of parallel tasks. It is clear that this functionality subsumes the current behaviour of `parallel`, and adds behaviour that was inexpressible before.

5. The final part that should be abstracted from is the arrangement, or layout, of the generated GUIs of the (control and regular) tasks. In the current iTask system a distinction is made between a parallel form for tasks that can, in principle, each be delegated to other workers and a parallel form for tasks which GUI should be merged into one single presentation. In order to abstract from this, it is better to parameterize the new parallel combinator with a function that describes how the component GUIs of (control and regular) tasks should merged.

We are currently experimenting with a single `parallel` combinator that meets the above criteria. With this combinator we hope to express all other task combinators as special cases. This should aid the development of a formal framework of the new iTask system. Note that, for efficiency reasons, an actual implementation may need to resort to specialized implementations.

5. CONCLUSIONS

The original iTask system offers a lot of functionality on a high level of abstraction liberating the programmer from worrying about many implementation details. The concept of an iTask task was a unit of work performed somewhere which, when finished, yielded a value of a certain type which is used to dynamically determine which other tasks to do next. With a fixed but small and powerful set of combinators complex work patterns can be captured.

In this paper we have argued that nevertheless more expressive power is needed. The purpose of the iTask library is not only to provide a concise formalism for defining tasks, but also to support the construction of WFMS's. From an iTask specification, an executable distributed web enabled WFMS is generated. The library should therefore have as little as possible predefined behaviour. In addition to the tasks that need to be done one also wants to be able to define the view and control managers have on the work that is going on. Furthermore, web browsers nowadays offer much more functionality than a couple of years ago. Instead of offering a simple form to be filled in, complete full-fledged GUI applications can be run in a web browser.

Currently we are redefining and re-implementing the iTask library. We had to change the tasks concept enabling a task to share information with others while the work is going on. In addition to regular tasks, special control tasks are added. Tasks can become a complete GUI application, offering buttons, menus, dialogues and multiple windows. Clearly, the iTask system is getting big.

At this stage we have not yet tested the new system with non-toy examples, but we hypothesize that we can capture *all* current iTask combinators *and* the above mentioned shortcomings with only *two* constructs: one very general `editor` and one Swiss-Army-Knife combinator for creating parallel tasks. These two should suffice to construct all other combinators.

Acknowledgements

This research is supported by the Dutch Technology Foundation STW, applied science division of NWO, and the Technology Program of the Ministry of Economic Affairs.

6. REFERENCES

[1] W. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. Technical Report FIT-TR-2002-02, Queensland University of Technology, 2002.

[2] A. Bohannon, B. C. Pierce, and J. A. Vaughan. Relational lenses: a language for updatable views. In *PODS '06: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 338–347, New York, NY, USA, 2006. ACM.

[3] G. Krasner and S. Pope. A cookbook for using the model-view-controller user interface paradigm in Smalltalk-80. *Journal of Object-Oriented Programming*, 1(3):26–49, Aug. 1988.

[4] S. Michels, R. Plasmeijer, and P. Achten. iTask as a new paradigm for building GUI applications. In J. Hage, editor, *Proceedings of the 22th'10, Selected Papers*, 2011. Accepted for publication.

[5] R. Plasmeijer, P. Achten, and P. Koopman. iTasks: executable specifications of interactive work flow systems for the web. In R. Hinze and N. Ramsey, editors, *Proceedings of the International Conference on Functional Programming, ICFP '07*, pages 141–152, Freiburg, Germany, 2007. ACM Press.

[6] R. Plasmeijer, P. Achten, P. Koopman, B. Lijnse, T. van Noort, and J. van Groningen. iTasks for a

change - Type-safe run-time change in dynamically evolving workflows. In S.-C. Khoo and J. Siek, editors, *Proceedings of the Workshop on Partial Evaluation and Program Manipulation, PEPM '11, Austin, TX, USA*. ACM Press, 2011.

[7] P. Wadler. Comprehending monads. In *Proceedings of the Conference on Lisp and Functional Programming, LFP '90, Nice, France*, pages 61–77, 1990.