iTasks for a Change

Type-Safe Run-Time Change in Dynamically Evolving Workflows

Rinus Plasmeijer[1]   Peter Achten[1]   Pieter Koopman[1]
Bas Lijnse[1,2]   Thomas van Noort[1]   John van Groningen[1]

[1] Institute for Computing and Information Sciences, Radboud University Nijmegen
P.O. Box 9010, 6500 GL, Nijmegen, The Netherlands

[2] Faculty of Military Sciences, Netherlands Defence Academy
P.O. Box 10000, 1780 CA, Den Helder, The Netherlands

{rinus, p.achten, pieter, b.lijnse, thomas, johnvg}@cs.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 based on a given workflow description. Due to changing circumstances, it happens often that some tasks in a running workflow need to be performed differently than originally planned and specified. Most commercial WFMSs cannot deal with the required run-time changes properly. These changes have to be specified at the level of the underlying Petri-Net based semantics. Moreover, the implicit external state has to be adapted to the new task as well. Such low-level updates can easily lead to wrong behaviour and other errors. This problem is known as the dynamic change bug. In the iTask WFMS, workflows are specified using a radically different approach: workflows are constructed in a compositional style, using pure functions and combinators as self-contained building blocks. This paper introduces a change concept for the iTask system where self-contained tasks can be replaced by other self-contained tasks, thereby preventing dynamic change bugs. The static and dynamic typing system furthermore guarantees that these tasks have compatible types.

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

General Terms   Design, Languages

Keywords   combinators, type-safe run-time change, workflow

1. Introduction

Workflow management systems (WFMS) are software systems that coordinate, generate, and monitor tasks performed by human workers and computers. Such systems are interesting to study because they are nontrivial representatives of how contemporary distributed software systems are manufactured. A concrete workflow ensures that essential actions are performed in the right order. Workflows have potentially long running times (in the order of months and

years) during which many workers and computers can be involved. It is well known from literature (Van der Aalst, 2001) that during its life cycle, a workflow needs to adapt to handle changing requirements, ad-hoc situations, evolutionary concerns, and so on. It also illustrates that adapting a running workflow system easily leads to incorrect behaviour. This phenomenon is known as the dynamic change bug (Ellis et al., 1995). Most traditional WFMSs are affected by this issue because their semantics is based on Petri Nets. Changes are made on the level of individual places and transitions, which is a too low level of abstraction. Making an arbitrary change in a Petri Net while the tokens are moving around often leads to errors. Moreover, the tasks have an implicit external state which is often stored in a database. This state has to be adapted to the changed task without corrupting the state of other tasks stored in the same database.

The iTask system (Plasmeijer et al., 2007) distinguishes itself from traditional WFMSs. First, iTask is actually a monadic combinator library in the pure and lazy functional programming language Clean. It defines a WFMS, embedded in Clean where the combinators are used to combine tasks. 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 (e.g., data storage and retrieval, GUI rendering, form interaction) because this is all dealt with using generic programming techniques. Fourth, 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 (i.e., tasks that create and accept other tasks) and recursion to the modelling repertoire of workflow engineers. Sixth, in contrast with the large catalogue of common workflow patterns (Van der Aalst et al., 2002), iTask workflows are captured by means of a small number of core combinator functions.

In this paper we show how type-safe run-time changes are incorporated in iTask. Together with the compositionality and self-containedness of tasks, this prevents dynamic change bugs. Type-safe run-time change is challenging for several reasons. First, an iTask workflow dynamically evolves. This implies that the set of

edit		:: String a	→ Task a	iTask a
(@:)	**infixr** 5 :: p (Task a)		→ Task a	property p & iTask a
return		:: a	→ Task a	iTask a
(≫=)	**infixl** 1 :: (Task a) (a → Task b)		→ Task b	iTask b
(–‖–)	**infixr** 3 :: (Task a) (Task a)		→ Task a	iTask a
(-&&-)	**infixr** 4 :: (Task a) (Task b)		→ Task (a, b)	iTask a & iTask b

Figure 1. The iTask core combinator functions

tasks that may require replacement is not known in advance and that a change must take both the current workflow state into account as well as its future. Second, all tasks are typed and include context restrictions for ad-hoc polymorphic and generic functions. This implies that a single change is really about changing a set of functions. Third, iTask workflows are constructed using combinators. We show that we can handle all changes by modifying only one single combinator drastically. For the sake of clarity, we emphasize that in this paper we concentrate on the change mechanism and its semantics. There are still many challenging research questions to be answered with respect to validation and formal reasoning of changes, but this remains future work.

This paper is organized as follows. First, we describe the core iTask system and explain its usage by means of a running example: a paper review workflow (Section 2). Next we explain the concept of change functions and illustrate how it is used to alter the paper review workflow (Section 3). We capture the semantics by first giving a reference implementation of the core combinator functions (Section 4) after which we extend it with type-safe run-time change (Section 5). Finally, we compare our approach with related work (Section 6) and conclude and discuss future work (Section 7).

2. The iTask core system

In this section we give a brief overview of the iTask system. We restrict ourselves, without loss of generality, to the core combinator functions (Section 2.1), which we then use in the running example (Section 2.2). The actual iTask system offers a couple of additional combinators for advanced task structures, workflow process management, and exception handling. The change mechanism is orthogonal to these features.

2.1 The core combinator functions

The core combinator functions of the iTask system are displayed in Figure 1. (Note that in Clean the arity of functions is shown explicitly by separating argument types by spaces instead of →.) A task has an opaque type Task a: the type parameter a is the type of the result value that is committed once the task finishes.

One primitive task concerns editors: an editor is created with edit prompt va. When applied to an initial va of 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 generic functions (hence the context restriction | iTask a) which is derived by the compiler automatically. The iTask system guarantees that only values of type a are created. This continues until the worker decides to commit the value to the workflow, which terminates the task. The prompt argument provides the worker with information about the purpose of this task.

Tasks are assigned properties using the @:combinator. For now, we restrict the properties to an identification label and the user who has to work on the task (of the opaque types Label and User respectively). If u :: User and l :: Label, then u @: ta, l @: ta, and (u, l) @: ta sets the worker, label, and both respectively of the task

ta. In general, properties that can be set are captured by the type class property, which is discussed later in Section 3.1. We call tasks annotated using the @:combinator, *main tasks*. As we will see later on, such tasks are subject to change once the workflow is running.

To compose tasks sequentially, the monadic combinators return and ≫= (Wadler, 1990) are provided. The task return va succeeds immediately and emits its value va. In ta ≫= atb, the task ta is evaluated first. When this task returns its value, say va, it is passed to atb 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.

2.2 Running example: a paper review workflow

In this section we illustrate the use of the iTask system by designing a workflow for reviewing papers that can be part of a conference management system.

We start by defining the types that determine the universe of discourse. For conciseness, we use very simple types: a paper is represented by some opaque type Paper for which two access functions are available to display the title and full content:

```
:: Paper
title :: Paper → String
full  :: Paper → String
```

We define two label-generating functions, using a constructor function label :: String → Label, to identify the tasks for reviewing and bidding for papers:

```
reviewLabel :: Paper → Label
reviewLabel p = label ("review " + title p)

bidLabel :: Paper → Label
bidLabel p = label ("bid " + title p)
```

The chair and other members are synonyms of the opaque type User:

```
:: Chair    :== User
:: PC       :== [Reviewer]
:: Reviewer:== User
```

Programme committee members bid for papers to review, and for those papers that they are in charge of, they make a decision. A decision is either to Reject, Discuss, or Accept a paper:

```
:: Bid      :== (Reviewer, [Paper])
:: Review   :== (Paper, (Reviewer, Decision))
:: Decision  = Reject | Discuss | Accept
```

This settles the universe, we proceed with the tasks. First we design a task that lets all programme committee members bid for papers. We proceed in a bottom-up fashion and first create another main task to ask a programme committee member whether she would like to review a specific paper (the value True is the default value):

```
choosePaper :: Paper → Task Bool
choosePaper p = bidLabel p @: edit (title p) True
```

We present the choice for each paper and obtain the selected papers using a parallel list comprehension that filters the chosen papers:

```
choosePapers :: [Paper] → Task [Paper]
choosePapers ps = all (map choosePaper ps) ≫= λds →
                return [p \\ p ← ps & True ← ds]
```

The function all guarantees that all tasks in the list are evaluated to completion, and collects and returns their results:

```
all :: ([Task a] → Task [a]) | iTask a
all = foldr (λta tas → ta -&&- tas ≫= λ(va, vas) →
                    return [va : vas]) (return [])
```

Next, we request the bid of all programme committee members:

```
assignPapers :: PC [Paper] → Task [Bid]
assignPapers pc ps = all [ member @: (choosePapers ps ≫= λs →
                                          return (member, s))
                           \\ member ← pc
                         ]
```

The resulting list tells us which papers the programme committee members want to review. In a real system, the chair verifies the number of reviewers per paper, which is not modelled here due to limited space.

Now we design a single paper review for some reviewer, identified by the title of the paper. The reviewer either reviews the paper or delegates the task to another reviewer. Hence, this task is modeled as a choice between two alternatives:

```
reviewPaper :: Paper Reviewer → Task Review
reviewPaper p r = reviewLabel p @:
                    (plainReview p r –||– delegateReview p r)
```

A plain review of a paper shows the paper to the reviewer, who decides to reject, discuss (the initial value), or accept the paper:

```
plainReview :: Paper Reviewer → Task Review
plainReview p r = edit (full p) Discuss ≫= λd →
                    return (p, (r, d))
```

Alternatively, a reviewer may decide to delegate the review of a paper to another reviewer. Hence, reviewPaper is an example of a recursive workflow. The original reviewer remains responsible for reviewing the paper, which explains why she is still passed as an argument to reviewPaper:

```
delegateReview :: Paper Reviewer → Task Review
delegateReview p r = edit "Delegate review" r ≫= λother →
                       other @: reviewPaper p r
```

Given this assignment, we ask all members to perform their tasks:

```
reviewPapers :: [Bid] → Task [Review]
reviewPapers bids = all [ r @: reviewPaper p r
                          \\ (r, ps) ← bids, p ← ps
                        ]
```

The entire workflow of first making a selection of papers, followed by reviewing all papers by all reviewers, is a simple succession:

```
reviews :: PC [Paper] → Task [Review]
reviews pc ps = assignPapers pc ps ≫= reviewPapers
```

A conference management system can use this result to automatically separate all clear cases (i.e., all reject and all accept) from the papers that require discussion.

The final step is to turn this specification into an executable application. The main function in Clean is called Start and must accept and return a unique *World value in order to interact with the external world. The type attribute * is due to Clean's uniqueness typing (Barendsen and Smetsers, 1993) to ensure single-threaded use of the corresponding value, which allows us to model side-effects in a pure functional language. The library function startEngine takes a list of workflow specifications, using the constructor function workflow that are made available to the workers:

```
Start :: *World → *World
Start world = startEngine [workflow "Review" reviewWF] world

reviewWF :: Task [Review]
reviewWF = getUsersWithRole "chair"    ≫= λchairs →
             getUsersWithRole "reviewer" ≫= λpc     →
             readDB "papers"              ≫= λpapers →
             (hd chairs, label "reviews") @: reviews pc papers
```

To complete the example, the proper workers and papers need to be loaded. The iTask system supports several functions to obtain information about its workers. The library function getUsersWithRole takes a role and yields all currently registered workers who have that role, readDB takes an identifier and reads the information from the corresponding database. Finally, we assign the main review tasks reviews to the chair with the label "reviews".

3. Type-safe run-time change

The iTask system includes many features to define workflows. Despite this expressive power, it is cumbersome to anticipate on every possible future way of working in a workflow specification. Moreover, it is too much work to anticipate on all changes that can happen simply because the workflow would become much too complicated. The ability to make changes at run time helps to keep workflow specifications concise. When specifying the workflow, we define the current way of working. At run time it is determined when and where a change in the way of working is needed.

In order to keep the system simple, we only allow main tasks to be changed, meaning, tasks explicitly labelled using the @: combinator. This is not a fundamental restriction since any task can be promoted to a main task. Moreover, other combinators can be made aware of changes in the same fashion.

In the remainder of this section we first explain what properties of a task are (Section 3.1) and what a change function is (Section 3.2). We create an API for often occurring change patterns (Section 3.3). Next we use this API in the running example (Section 3.4) to show a number of changes.

3.1 Properties

As alluded to in Section 2.1, tasks can have several properties. These are captured by the opaque type Properties. The type class property defines the properties that can be set and get:

```
class property p where
  setProperty :: p Properties → Properties
  getProperty :: Properties → p

instance property Label, User
```

Instances for the types Label and User are used in this paper.

3.2 Dynamic change

A workflow under execution consists at any time of at least one main task. The mechanism of change that we propose in this paper is as follows. A change is a function that is applied to a main task if it fits the properties and type of that task. Based on the properties of the main task to be changed, the change function decides whether or not to change the properties, the entire main task, and also how to continue changing in the future. This is performed for all current main tasks, as well as for the future main tasks in the workflow. Hence, a change that alters main tasks of type Task a is a function of type Change a:

```
:: Change a :== Properties (Task a) (Task a)
               → ( Maybe Properties
                 , Maybe (Task a)
                 , Maybe ChangeContinuation
                 )
```

Its arguments are the current properties p of the main task and expressions corresponding to the current task current and the original description of the task original, both of type Task a.

If the change function yields new properties np, these are used instead of the current properties p. Such a change establishes for instance that a specific task is moved from one worker to another.

If the change function yields a new task new, the old task current is aborted and replaced by new. Naturally, new has to be of the

same type as current. It can be constructed from the task current, representing the task possibly under current evaluation, as well as the original task description original, representing the same task before people were working on it. So, we can either construct a complete new task, or continue with the work but let it for example be followed by a supervising task inspecting its result, or restart the main task from scratch if necessary. By changing the task, a worker might be faced with the fact that her work is no longer needed or that something else has to be performed. The iTask system informs the user about such issues.

If the change function yields a continuation cf, the type of which is discussed below, then the change function wants to alter another main task, but now it will use the change function cf. This allows the change function to alter its behaviour, making use of the knowledge it obtained from inspecting the main tasks so far. When the change function yields no continuation, the change is said to be *exhausted*. After changes have been applied to every current main task, any remaining continuations are stored in order such that they can be applied to future main tasks. Stored continuations are removed automatically when the corresponding change is exhausted, or manually by an authorized user.

The type of the ChangeContinuation deserves special attention. The type Change a is too restrictive since only main tasks of the same type can be changed. Returning a change of type ∀ a: Change a is too liberal, with this type we cannot limit changes to tasks of a specific type. The dynamic typing system of Clean (Pil, 1999) solves this problem by defining a change continuation as a dynamic type:

```
:: ChangeContinuation:== Dynamic
```

The dynamic value contains a change function of some type Change a. The iTask system can only apply this function to a main task of type Task b if the types a and b can be unified at run time, as we will see in Section 5.3. Changes of a type that cannot be unified with a main task are ignored.

3.3 Change combinators

For convenience, we define a set of change combinators to create proper change functions, as shown in Figure 2. We give examples of their use in Section 3.4.

The initial change change0 terminates change handling immediately and alters neither properties nor tasks. Changes that only concern properties, tasks, or the change continuation are expressed preferably as functions over Properties, Task, and Change functions. These functions are lifted to change functions with newProps, newTask, and newChange respectively. In the definition of newChange we have to make sure that the provided function over change functions fits the value stored in the change continuation. We use a type-dependent dynamic pattern match (Pil, 1999) with the type attribute ^ to check that their types unify. Another useful variant is forceChange ncf, which forces the change continuation to always become ncf; even if the current change is exhausted.

Switching between two alternative change functions tf and ef is achieved by switch cond tf ef, using a predicate cond on the properties of the main task. Two frequently occurring patterns are expressed in terms of switch: the function when cond cf applies its change only when cond holds, and once cond cf applies its change exactly once to the first task that satisfies cond and terminates.

A change function is applied to exactly one task, and then determines how to proceed by returning a change continuation function. Iteration of the same change is captured with repeat cf, which makes sure that cf is applied from now on.

Sometimes it is desirable to overrule the result of a main task, depending on its current properties. We introduce two overruling functions: overruleOne cond vf overrules the first task that satisfies the predicate cond, and overruleAll cond vf overrules all of them.

3.4 Running example: changing the paper review workflow

We use the change combinators of Figure 2 to construct change functions for the paper reviewing workflow example.

Suppose that reviewer r1 is no longer able to perform her duties, and the symposium chair decides that all her work must be handed over to reviewer r2. This is realized by the change function handOver:

```
handOver :: Reviewer Reviewer → Change a | iTask a
handOver r1 r2 = repeat (delegate r1 r2)

delegate :: Reviewer Reviewer → Change a | iTask a
delegate r1 r2 = newProps changeProp change0
where changeProp p | getProperty p == r1 = setProperty r2 p
                   | otherwise           = p
```

A slightly more involved example is to delegate the work of the member to a list of members $[pc_1 .. pc_n]$ in round-robin order:

```
distribute :: Reviewer [Reviewer] → Change a | iTask a
distribute r []      = change0
distribute r [pc:pcs] = forceChange (distribute r (pcs ++ [pc]))
                                    (delegate r pc)
```

As another example, suppose that the symposium chair decides that a reviewing task has to be supervised by someone identified as super. The result of a completed task ta is offered to super, who decides to edit this result and commit it or redo the entire task:

```
check :: String Reviewer (Task a) (Task a) → Task a | iTask a
check s super ta t0 = ta ≫ λva → super @: (edit s va −||− t0)
```

The symposium chair can now conditionally supervise tasks:

```
supervise :: (Properties → Bool) String Reviewer → Change a
                                                  | iTask a
supervise cond s super = repeat (when cond (newTask (check s super)
                                                    change0))
```

Such kind of overruling can be performed to the extreme: suppose that the symposium chair decides that the result of a certain main task (e.g., a review) must be a specific value. If the chair knows the label of the task, then this action is formalized as:

```
overrule :: Label a → Change a | iTask a
overrule l x = overruleOne (λps → getProperty ps == l) (const x)
```

Suppose that a late paper p arrives and the programme chair decides to enter this paper in the reviewing workflow. She needs to extend the main workflow, identified by the label "reviews" as used in Section 2.2, with a new instance of an entire review for the single paper p:

```
late :: PC Paper → Change [Review]
late pc p = once (λps → getProperty ps == label "reviews") add
where add = newTask
              (λta _ → ta −&&− reviews pc [p] ≫ λ(old, new) →
                      return (old ++ new))
            change0
```

Conversely, suppose that an author withdraws her paper. In that case, programme committee members should stop bidding or reviewing that particular paper. Such tasks are identified by their bidLabel and reviewLabel:

```
noBid :: Paper → Change Bool
noBid p = overruleAll (λps → getProperty ps == bidLabel p)
                      (const False)

noReview :: Paper → Change Review
noReview p = overruleAll (λps → getProperty ps == reviewLabel p)
                         (λps → (p, (getProperty ps, Reject)))
```

After the manager has applied these two change functions to the workflow, every bid is converted to not selecting it, and a review is

```
change0                 :: Change a
change0                 = λp ta t0 → (Nothing, Nothing, Nothing)

newProps                :: (Properties → Properties) (Change a) → Change a        | iTask a
newProps fp cf          = λp ta t0 → case cf p ta t0 of (Nothing, nta, ncf) → (Just (fp  p), nta, ncf)
                                                        (Just np, nta, ncf) → (Just (fp np), nta, ncf)

newTask                 :: ((Task a) (Task a) → Task a) (Change a) → Change a     | iTask a
newTask fta cf          = λp ta t0 → case cf p ta t0 of (np, Nothing,  ncf) → (np, Just (fta  ta t0), ncf)
                                                        (np, Just nta, ncf) → (np, Just (fta nta t0), ncf)

newChange               :: ((Change a) → Change a) (Change a) → Change a          | iTask a
newChange fcf cf        = λp ta t0 → case cf p ta t0 of (np, nta, Just (ncf :: Change a^)) → (np, nta, Just (dynamic (fcf ncf)))
                                                        (np, nta, ncf)                     → (np, nta, ncf)

forceChange             :: (Change a) (Change a) → Change a                       | iTask a
forceChange ncf cf      = λp ta t0 → case cf p ta t0 of (np, nta, _) → (np, nta, Just (dynamic ncf))

switch                  :: (Properties → Bool) (Change a) (Change a) → Change a | iTask a
switch cond tf ef       = λp ta t0 → if (cond p) (tf p ta t0) (ef p t0)

when                    :: (Properties → Bool) (Change a) → Change a              | iTask a
when cond cf            = switch cond cf change0

once                    :: (Properties → Bool) (Change a) → Change a              | iTask a
once cond cf            = switch cond cf (forceChange (once cond cf) change0)

repeat                  :: (Change a) → Change a                                  | iTask a
repeat cf               = forceChange (repeat cf) cf

overruleOne             :: (Properties → Bool) (Properties → a) → Change a        | iTask a
overruleOne cond vf = once cond (λp → newTask (λ_ _ → return (vf p)) change0 p)

overruleAll             :: (Properties → Bool) (Properties → a) → Change a        | iTask a
overruleAll cond vf = repeat (when cond (λp → newTask (λ_ _ → return (vf p)) change0 p))
```

Figure 2. The iTask change combinators

changed to reject it. As a consequence, the programme committee members do not perform redundant work.

The above functions are all examples of change functions. They can be stored and loaded as dynamic values in arbitrary Clean programs. To actually apply such a change function to an iTask workflow, an authorized user loads such a dynamic change function, identifies it with a label, and adds it to the workflow. The label is required to allow the user to remove the change function at any later time.

With these examples, we demonstrate that realistic changes can be modelled concisely with a set of combinators. Each of these changes could have been anticipated within the original workflow, but this would have resulted in an unwieldy workflow specification.

4. The iTask core reference implementation

In this section we first capture the semantics of the iTask core system as defined in Section 2. Change handling as described in Section 3, is treated in Section 5. As we will see, change handling is defined in such a way that it only affects the @: combinator. The semantic definitions of all other combinators remain the same.

We express the semantics in terms of a reference implementation, using Clean as modeling language. There are several advantages to this approach. First, using a strongly typed programming language with an expressive type system gives static type checking of the semantic model for free. Second, it allows the model to be executed to verify if it behaves as expected. Execution of a carefully designed set of unit tests appeared to be crucial in order to validate various versions of the semantics. Third, use of Clean creates no artificial gap between the modeling language and actual

implementation, which would occur had we used Agda, Coq, or other means.

The semantic description we present here for the iTask core system is different from an earlier version (Koopman et al., 2009). The main goal of that work was to establish an initial semantics and investigate its properties using model-based testing. In that approach, we make use of an algebraic datatype to model combinators and are forced to use a closed universe of task types. Here we define the semantics of the combinators using semantic functions; there are no restrictions on the types being used. Instead of semantic functions, the use of generalised algebraic data types (Peyton Jones et al., 2006) would have been an alternative approach, but this requires an integration with dynamic types (Van Noort et al., 2010a).

The signatures of the semantic functions correspond closely to the original signatures of the modeled combinator functions. The semantic function of a combinator function of type Task a has type STask a (Section 4.1) and describes a one-step reduction of a task given a worker event and current state of the workflow.

In the reference implementation we abstract from all details that involve boilerplate programming, such as data storage and retrieval, GUI rendering, and form interaction. For this reason, the semantic functions do not include the iTask context restriction, but do require Dynamic to handle input of arbitrary type.

Although workers are assumed to work in a distributed setting, the events they produce are collected and sequentially offered to the iTask system. It handles events one by one in a single event-handling loop (Section 4.2) such that we do not have to worry about concurrency. The reduction of tasks with respect to event sequences describes a term-rewriting system. It is a fixed-point computation

that eventually may lead to a task in normal form. The effect of one reduction step is a reduced term which, if the fixed point is not reached, is capable of handling the next event. Additionally, the set of next possible input events is calculated, which is used by the real iTask system to generate proper feedback and a GUI for the workers.

4.1 Semantic types

A semantic function performs a one-step reduction based on the current identification and properties of the task, as well as the worker event and state of the system. Its type is defined as follows:

```
:: STask a:== TaskId Properties Event State → ∗(Reduct a, State)
```

As a notational convention, semantic functions use the formal parameters λ i p e s. Next, we explain the individual components of the STask type.

Stable task identifiers   Several workers can work on tasks at the same time. Since each worker has her own GUI, the state of the workflow system can be changed by someone else while a worker is working on a task. Meanwhile, new tasks can be created while others are no longer needed. Hence, we have to guarantee that a task someone is working on is uniquely identified and that this identification remains the same over time under all conditions. To enforce such stable unique task identifiers, we impose an identification scheme that encodes the path from the root of the task expression to the given combinator function by a list of integers:

```
:: TaskId:== [Int]

taskId0 :: TaskId
taskId0 = [0]
```

We use taskId0 as the initial TaskId. A combinator with unique identification i identifies its subexpressions uniquely as [0 : i], [1 : i], and so on, as specified by subIds:

```
subIds :: TaskId → [TaskId]
subIds i = [[n : i] \\ n←[0 .. ]]
```

Properties   As described earlier in Section 3.1, tasks can have several properties. Hence, the semantic function of a task takes such properties. The initial properties are defined by props0, which we do not define in this paper since the Properties type is opaque.

Events   The generic foundation of the iTask system permits us to abstract from GUI programming, and instead refer to GUI elements in terms of the value of their editors. The value of an editor can be of any type, hence we use a dynamic to store its value.

A worker who terminates an editor, emits a Commit tid event. A worker who manipulates the GUI that is generated by an edit task with identification value tid, updates a new value of the corresponding type and emits an Update tid (**dynamic** value) event. Not all worker events are related to editor tasks: examples of other events are when a worker signs into the system or requests to refresh (part of) the GUI:

```
:: Event   = Commit TaskId | Update TaskId Value | Refresh
:: Value:== Dynamic
```

Reductions of subtasks (e.g., return) can cause further reduction which needs to be triggered as well. We model this behaviour also using a Refresh event.

State   The combinator library of the iTask system is a monadic state transformer. The unique state consists of two components: the values of all currently active, nonterminated editors, and the unique external world that links the iTask workflow with its environment:

```
:: ∗State       = {es :: [EditorValue], world :: ∗World}
:: EditorValue:== (TaskId, User, Value)
```

Note that by including the type attribute ∗ in the definition of State, any occurrence of that type is attributed unique automatically.

While handling an event, every nonterminated editor accumulates its current value in the collection of editor values. As a result, after the workflow has entirely handled the event, the state has an exact and complete record of the values of all currently active editors. The presence of the external world in the state demonstrates that the workflow engineer can, in principle, create custom tasks with side-effects. It should be noted that we do not discuss this further in this paper. In addition, it makes the connection between the workflow and the external world explicit, using the following two functions:

```
showGUI      :: State → State
getNextEvent :: State → (Event, State)
```

The function showGUI renders the values of the current active editors (which are collected in the es field of the state), and displays them in the external world. The function getNextEvent retrieves an event from the external world. Their definitions are not relevant to this paper. Finally, the initial state is created with state0:

```
state0 :: ∗World → State
state0 world = {world = world, es = []}
```

Reducts   The reduction semantics describes the reduction behaviour for each combinator of the iTask system with a semantic function. A task combinator expression reduces to either task normal form or to a new task combinator expression:

```
:: Reduct a = NF a | Redex (STask a)
```

A task is in normal form when it is terminated and has committed its final value; its reduct is NF value. Note that the committed value is lazy like any other value in Clean and hence not necessarily in normal form. The reduct of a task that is not yet terminated, is a new task expression. Such a value nta has type STask a, and the reduct has value Redex nta.

4.2 Task normalization

The evaluation of a combinator expression is defined by the function normalize which is a fixed-point computation that terminates when a normal form is reached:

```
1  normalize :: (STask a) ∗World → (a, ∗World)
2  normalize ta world = toNF (startTask ta Refresh (state0 world))
3
4  toNF :: (Reduct a, State) → (a, ∗World)
5  toNF (NF va,   s) = (va, s.world)
6  toNF (Redex ta, s) = toNF (reduce ta (getNextEvent (showGUI s)))
7
8  reduce :: (STask a) (Event, State) → (Reduct a, State)
9  reduce ta (e, s)
10   | valid e s.es = startTask ta e s
11   | otherwise   = (Redex ta, showError "Invalid event" e s)
12
13 startTask :: (STask a) Event State → (Reduct a, State)
14 startTask ta e s = ta taskId0 props0 e {s & es = []}
```

All current editor values that are accumulated in the state are offered to the workers using showGUI. They either update or commit an arbitrary editor with an arbitrary (but type-preserving) new value which generates an event that is collected using getNextEvent. Only valid events cause a reduction of the workflow (line 10). When evaluating a new event, the task is provided with the proper initial values of task identification, properties, and an empty accumulator of editor values, as defined by startTask (lines 13-14).

Invalid events can be generated by workers who are looking at old versions of the workflow in their GUI and generate events corresponding to tasks which are no longer needed. Due to the consistent naming of tasks, we can determine if an event belongs to an editor that is still active by comparing its identifier to the identifiers of active editors collected in the state:

```
valid :: Event [EditorValue] → Bool
valid (Commit tid)   es = isMember tid [ j \\ (j, _, _) ← es ]
valid (Update tid _) es = isMember tid [ j \\ (j, _, _) ← es ]
valid _              _  = True
```

In case of an invalid event, an error message is shown using the helper function showError (line 11).

4.3 Combinator semantics

We now define the semantic functions for the iTask core system shown in Figure 1. Each occurrence of Task is replaced by STask in the original signatures.

We start with the edit combinator, which is the only semantic function that inspects Commit and Update events:

```
1  edit :: String a → STask a | Dynamic a
2  edit _ va = λi p e s →
3    case e of
4      Commit tid
5        | tid == i → (NF va, s)
6      Update tid (nva :: a^)
7        | tid == i → (Redex (edit nva), addES i p nva s)
8        _          → (Redex (edit va),  addES i p va s)
9
10 addES :: TaskId Properties a State → State | Dynamic a
11 addES i p va s = {s & es = [(i, getProperty p, dynamic va) : s.es]}
```

An event matches an editor when the task identifications values are equal (lines 5 and 7). Although we know ourselves that the type of an updated value is identical to the type of the value of an editor, we need to guarantee this using a type-dependent dynamic pattern match (line 6), as seen earlier in Section 3.3. The three kinds of events, as defined in Section 4.1, each inflict different behaviour of edit. First, if the worker has terminated an editor task, it reduces to normal form with the current value stored in the editor (line 5). Second, if the worker has changed the current value of the editor task to nva (line 6), then it reduces to an editor task with the nva value (line 7). In any other case the current value of the editor is not changed (line 8). Except when the task normal form is reached, the current value is accumulated in the workflow state.

The semantic function of @: assigns a new property to the given task. This is achieved straightforwardly by updating the old properties with the new property as long as this combinator is a redex:

```
(@:) infixr 5 :: p (STask a) → STask a | property p
(@:) np ta  = λi p e s →
  case ta i (setProperty np p) e s of
    (Redex nta, s) → (Redex (np @: nta), s)
    reduct         → reduct
```

The new properties set by np @: ta are inherited by all subtasks within ta until overruled by another @: combinator.

The monadic combinators are return and ≫= :

```
1  return :: a → STask a
2  return va = λi p e s → (NF va, s)
3
4  (≫=) infixl 1 :: (STask a) (a → STask b) → STask b
5  (≫=) ta atb = λi p e s →
6    case ta (subIds i !! 0) p e s of
7      (NF va,     s) → setTaskId (λi → subIds i !! 1) (atb va)
8                                  i p Refresh s
9      (Redex nta, s) → (Redex (nta ≫= atb), s)
```

The semantic function of return immediately emits its argument value as normal form without modifying the state (line 2). The semantic function of ≫= is the bind operator of a state monad. In ta ≫= atb, the task ta is reduced one step first (line 6). (The expression xs !! i selects the ith element of xs.) If a task normal form is reached, then the committed value va of that task is provided to the second argument of ≫= (line 7). A Refresh event is applied to the newly created task in order to collect its events in the state (line 8). If the task is still a redex, then a new redex is constructed (line 9). Note that caution is advised in numbering subexpressions since the one-step reduction can change the shape by transforming ta ≫= atb to atb va, where va is the task normal form value of ta. The function setTaskId is used to warrant the correct identification of new subtasks:

```
setTaskId :: (TaskId → TaskId) (STask a) → STask a
setTaskId fi ta = λi p e s →
  case ta (fi i) p e s of
    (Redex nta, s) → (Redex (setTaskId fi nta), s)
    reduct         → reduct
```

When given a function that generates a correct task identification value, setTaskId will always start numbering with this identification value. In case of ≫=, the second subtask needs to receive the second subidentification value at index 1 (line 7).

The semantic function of –||– is defined as follows:

```
1  (–||–) infixr 3 :: (STask a) (STask a) → STask a
2  (–||–) ta ua = λi p e s →
3    case ta (subIds i !! 0) p e s of
4      (NF va,     s) → (NF va, s)
5      (Redex nta, s) →
6        case ua (subIds i !! 1) p e s of
7          (NF wa,     s) → (NF wa, s)
8          (Redex nua, s) → (Redex (nta –||– nua), s)
```

When reduction of the left subtask ta yields a normal form, the corresponding value is the result of this expression (line 4). Otherwise the right task, ua, is reduced. If that yields a normal form the entire reduction is also finished (line 7). When neither subtask is finished a new instance of the combinator is constructed using the updated subtasks (line 8).

The semantic function of –&&– is defined similarly:

```
1  (–&&–) infixr 4 :: (STask a) (STask b) → STask (a, b)
2  (–&&–) ta tb = λi p e s →
3    let (reducta, s1)  = ta (subIds i !! 0) p e s
4        (reductb, s2)  = tb (subIds i !! 1) p e s1
5    in case (reducta, reductb) of
6        (NF va,     NF vb   ) → (NF (va, vb),                s2)
7        (NF va,     Redex ntb) → (Redex (return va –&&– ntb), s2)
8        (Redex nta, NF vb   ) → (Redex (nta –&&– return vb), s2)
9        (Redex nta, Redex ntb) → (Redex (nta –&&– ntb),       s2)
```

First, the left subtask, ta, is reduced. Then, the right subtask, tb is reduced using the state of the first reduction. When both tasks are in normal form, reduction is finished and the results are paired and returned (line 6). Otherwise, a new instance of the combinator is constructed using the updated tasks (lines 7-9). If only one of the subtasks is in normal form, return is used to create a trivial redex.

5. Change handling reference implementation

In this section, we discuss what it takes to extend the reference implementation of the iTask core system with change handling. We first describe the changes to the elements of the semantic types (Section 5.1). We then proceed with the normalization of tasks (Section 5.2) and show how changes that enter a workflow during reduction are added and can be removed. Handling current and pending changes is performed only by the change-aware @: combinator (Section 5.3).

5.1 Semantic types

The type of a semantic function remains the same (we repeat it for easy reference), only the types for events and state are altered:

```
:: STask a:== TaskId Properties Event State → *(Reduct a, State)
```

Events As discussed in Section 3.4, authorized users can add and remove changes. This is captured by two new worker events to add (AddChange) and remove (RemoveChange) changes:

```
:: Event          = Commit TaskId | Update TaskId Value | Refresh
                  | AddChange LabeledChange
                  | RemoveChange (ChangeLabel → Bool)
:: LabeledChange:== (ChangeLabel, ChangeContinuation)
:: ChangeLabel   :== String
```

State The state is extended with change administration:

```
:: *State    = { es :: [EditorValue], world :: *World
               , cc :: Maybe LabeledChange
               , cq :: Queue LabeledChange
               }
:: Queue a:== [a]
```

Changes are stored in the state for two reasons. First, a change function modifies itself via its continuation, which is stored in the cc field of the state. Second, when all current main tasks have been changed and we still have a continuation, then the change needs to be applied to future main tasks. This change is appended to the change queue cq. Whenever new tasks are created this queue of change functions is applied in the same order as they have been issued. We define several access functions:

```
setChange :: (Maybe LabeledChange) State → State
setChange nc s = {s & cc = nc}

queueChange :: (Maybe LabeledChange) State → State
queueChange (Just c) s = {s & cq = s.cq ++ [c]}
queueChange _          s = s

storeChange :: State → State
storeChange s=:{cc} = setChange Nothing (queueChange cc s)
```

The function setChange updates the current change of the state, queueChange appends a change function to the queue of pending changes, and storeChange moves the current change to the queue of pending changes.

5.2 Task normalization

As we have seen in Section 4.2, a running workflow is characterized by the semantic functions normalize and reduce. We only need to adapt reduce to add and remove changes:

```
1  reduce :: (STask a) (Event, State) → (Reduct a, State)
2  reduce ta (e, s) =
3    case e of
4      AddChange c
5        → let (reduct, s2) = startTask ta Refresh
6                                      (setChange (Just c) s)
7          in  (reduct, storeChange s2)
8      RemoveChange p
9        → (Redex ta, {s & cq = filter (not o p o fst) s.cq})
10   _ | valid e s.es = startTask ta e s
11     | otherwise    = (Redex ta, showError "Invalid event" e s)
```

Nonchange events are handled exactly as before (lines 10-11). Adding a change causes reduction of the task with the given change set as current change (lines 5-6). After this reduction, the (potentially updated) change is stored in the queue of pending changes (line 7). Removing a change is a matter of keeping changes that do not satisfy the removal predicate p (lines 8-9).

5.3 Change-handling semantics

The key challenge is to adapt the @: combinator to make it aware of changes. This is a complex operation for several reasons. First, when np @: t0 is reduced, t0 is the initial subtask description that is required as third argument of any matching change function. As a consequence, t0 needs to be memorized during further reductions. Second, as soon as a new main task is created, all pending changes need to be applied to it in the order of their occurrence. When the task changes, we change the identification value of the subtask (by incrementing its index in the list of subidentification values that is generated with subIds, see Section 4.1). Third, any matching change produces either a continuation which should be used next time or it is exhausted.

For clarity of presentation, we have collected the necessary definitions in Figure 3. In contrast to the earlier semantic function definitions, the reduction behaviour of the @: combinator that includes changes consists of two stages. First, all pending changes are handled exactly once (line 3). Naturally, if this results in a task in task normal form, we are done (line 5). Otherwise, the second step is to reduce to the internal combinator ((nn, np, t0) @:+ nta) (line 4). It keeps track of the correct subtask identification index, properties, and the initial task, together with the result of the earlier reduction. The internal combinator @:+ keeps rewriting to itself as long as its subtask is not in task normal form (line 10). Note that the initial task t0 remains constant. If there is a current change, it is handled as described by doOneChange (line 12). If there is no current change, we only need to reduce its subtask (line 13).

Handling pending changes one by one amounts to folding doOneChange over the queue of pending changes. This is expressed concisely by doPendingChanges (line 18-19).

Finally, doOneChange is the pivotal function that actually applies a change to a task. It does not matter whether this change is a pending change or a new change. The only difference is that a pending change, if not exhausted, needs to be restored in the queue of pending changes, whereas a current change needs to be restored as current change. Hence, doOneChange is parameterized with a restore function of type RestoreChange (line 16) that restores the altered change function in the state. In case of @:+ this is the function setChange and in case of pending changes this is function queueChange (both functions were introduced earlier in Section 5.1). To make doOneChange suited for handling pending changes as well as current changes, it alters a structure of type ChangeResult a (line 15) which keeps track of the subtask identification index, properties, task (as a redex), and state. Required additional information is passed as the first three arguments: the initial task description, the current subtask identification index, and the means to restore the continuation in the state (line 22). A change can be applied without compromising type safety when either its type can be unified with the type of the task (line 24) or if the change function itself is sufficiently generic to handle the task (line 25). Both cases require extensions to standard dynamic typing: type-dependent dynamic pattern matching as seen earlier in Sections 3.3 and 4.3, and ad-hoc polymorphic functions in dynamic values (Van Noort et al., 2010b). If the checks fail, then we only need to restore the change function in the state (line 26). Handling a matching change is described by the change function (lines 27-32). We apply the change to the required arguments (line 29) and obtain the next subtask identification index, task, and change continuation (line 30). (The function fromMaybe removes the Just constructor from its second argument if it has one, and returns the first argument otherwise.) A new reduct is computed (line 32), the change continuation is restored, and possibly altered subtask identification index and properties are returned (line 27).

```
1   (@:) infixr 5 :: p (STask a) → STask a | property p & Dynamic a
2   (@:) np t0 = λi p e s →
3     case doPendingChanges t0 i (setProperty np p) s of
4       (nn, np, (Redex nta, s)) → ((nn, np, t0) @:+ nta) i np e s
5       (_ , _,  reduct)         → reduct
6
7   (@:+) infixr 5 :: (Int, Properties, STask a) (STask a) → STask a | Dynamic a
8   (@:+) (n, p, t0) ta = λi _ e s →
9     case doCurrentChange i e s of
10      (nn, np, (Redex nta, s)) → (Redex ((nn, np, t0) @:+ nta), s)
11      (_, _,  reduct)          → reduct
12  where doCurrentChange i Refresh s=:{cc = Just c} = doOneChange t0 i setChange (n, p, (Redex ta, {s & cc = Nothing})) c
13        doCurrentChange i e       s                 = (n, p, ta (subIds i !! n) p e s)
14
15  :: *ChangeResult a :== (Int, Properties, *(Reduct a, State))
16  :: RestoreChange   :== (Maybe LabeledChange) State → State
17
18  doPendingChanges :: (STask a) TaskId Properties State → ChangeResult a | Dynamic a
19  doPendingChanges t0 i np s=:{cq} = foldl (doOneChange t0 i queueChange) (0, np, (Redex t0, {s & cq = []})) cq
20
21  doOneChange :: (STask a) TaskId RestoreChange (ChangeResult a) LabeledChange → ChangeResult a | Dynamic a
22  doOneChange t0 i restoreChange (n, p, (r, s)) (ci, d) =
23    case (r, d) of
24      (Redex ta, cf :: Change a^)              → change ta cf
25      (Redex ta, cf ::∀ b: Change b | Dynamic b) → change ta cf
26      (reduct,   _)                            → (n,  p, (reduct, restoreChange (Just (ci, d)) s))
27  where change ta cf = (nn, np, (reduct, restoreChange nc s2))
28        where
29          (mbnp, mbnt, mbcf) = cf i p ta t0
30          (np, nta, nc)      = (fromMaybe p mbnp, fromMaybe ta mbnt, mapMaybe (λcf → (ci, cf)) mbcf)
31          nn                 = if (isJust mbnt) (n + 1) n
32          (reduct, s2)       = nta (subIds i !! nn) np Refresh s
```

Figure 3. The change-handling reference implementation of the @: combinator

6. Related work

There are three kinds of related work. First there are other WFMSs that recognized the need for dynamic changes of the workflow executed. Second, there are other constructs to change programs at run time. Finally, there are programming languages that have support for run-time change of code under execution.

The need for run-time changes in workflow systems has been recognized early (Ellis and Keddara, 2000; Ellis et al., 1995). These changes describe dynamic modification of both the specification of the workflow as well as any running instantiation of this specification. Several change patterns have been described (Weber et al., 2008), classifying types of changes that should be supported by a WFMS in two categories: adaption patterns and patterns for changes to predefined regions. The former describes changes to instantiations in which parts are either inserted, deleted, moved, replaced, and so on. The latter allows to defer the full specification of a workflow to run time, by predefining regions which are eligible to change at run time. Finally, in the untyped imperative ADEPT (Reichert and Dadam, 1998) workflow system, changes can only be applied if they satisfy preconditions imposed on global parameters or side-effects. Preconditions allow properties to be checked other than type correctness as in the iTask system. It is interesting to investigate how conditions can be added to the iTask system in general. Each of these approaches are confronted with the dynamic change bug. The Petri Nets representing the workflow is changed during its execution. Because these changes directly manipulate the structure of the Petri Net, this can quickly lead to inconsistencies. As the described reference implementation shows, a task is a pure and self-contained function which can only be replaced by another pure function of the same type.

The standard approach to changing behaviour at run time is known as the strategy pattern (Gamma et al., 1995). Here, the behaviour that can potentially be changed is insulated and the object possessing this behaviour is equipped with an explicit way to change it. In the context of this paper this would imply that we have to equip each task with such a hook. We believe our solution using change events that replaces a task by a new task is more elegant.

Finally, a number of programming languages offer support for run-time change of code. Erlang (Armstrong and Virdin, 1990) supports hot code swapping which is not type safe. ML (Duggan, 2001; Gilmore et al., 1997; Walton et al., 1998) offers type-safe dynamic module swapping. Imperative languages have been the subject of run-time change of code as well (Hicks, 2001; Stoyle et al., 2007). Aspect-oriented programming (Dantas et al., 2008; Kiczales et al., 1997) is another way to change programs dynamically. All these language-based approaches complicate the semantics significantly, our system possesses a plain rewrite semantics which we expect to make reasoning about changes considerably easier.

7. Conclusions and future work

In this paper we have demonstrated how running workflows can be changed type safely. This is a very important feature because deviations from the standard way of working is very common in daily practice. Yet, most commercial WFMS do not support making such changes properly and are subject to dynamic change bugs. This means that they are not of any help when their help is actually needed most. Adding the ability to perform type-safe run-time changes is difficult because their implementation commonly relies on Petri Nets and an implicit external state. Their focus is on control flow while the flow of information between tasks is realized as a side-effect storing information in databases.

In the iTask system, tasks are described by typed, pure, and self-contained functions which explicitly pass information to each other. Replacing a task means type-safe replacement of one pure function by another one. The type system ensures that the values passed between task have the correct type in the initial workflow as well as after any number of changes in this workflow.

Not only tasks under current execution can be changed type safely, also tasks generated in the future by the workflow can be changed. This is quite powerful, since tasks are evaluated dynamically and it is not known in advance which tasks come into existence. The change function can use what it has seen so far to decide how to act in the future.

We have captured the semantics of this powerful construct by giving a reference implementation in Clean. This enabled us to test its behavior. We have proven the practical applicability of the new feature by implementing it in the real iTask system, using the reference implementation as a lead. In the real system, tasks rely on the availability of generic functions. Hence we needed to be able to replace ad-hoc polymorphic functions in a type-safe fashion.

There are numerous possibilities for future work. For instance, the end user needs to be well informed about which work is not taken place as planned, which alternatives exist, and what their effects are. We want to investigate how the change concept can be used to resolve unexpected situations in the context of workflow support for crisis management. Since crisis situations are very unpredictable, it is essential that a WFMS is capable of run-time change in order to be of any use. Also, we plan to look at how we can reason more precisely about the effects of changes to a running workflow.

Acknowledgements

References

Wil van der Aalst. Exterminating the dynamic change bug: A concrete approach to support workflow change. *Information Systems Frontiers*, 3:3:297–317, 2001.

Wil van der Aalst, Arthur ter Hofstede, Bartek Kiepuszewski, and Ana Barros. Workflow patterns. Technical Report FIT-TR-2002-02, Queensland University of Technology, 2002.

Joe Armstrong and Robert Virdin. Erlang - An experimental telephony programming language. In *Proceedings of the International Switching Symposium, ISS '90, Stockholm, Sweden*, pages 2–7, 1990.

Erik Barendsen and Sjaak Smetsers. Conventional and uniqueness typing in graph rewrite systems (extended abstract). In Rudrapatna Shyamasundar, editor, *Proceedings of the Conference on the Foundations of Software Technology and Theoretical Computer Science, FSTTCS '93, Bombay, India*, volume 761 of *Lecture Notes in Computer Science*, pages 41–51. Springer-Verlag, 1993.

Daniel Dantas, David Walker, Geoffrey Washburn, and Stephanie Weirich. Aspectml: a polymorphic aspect-oriented functional programming language. *ACM Transactions on Programming Languages and Systems*, 30 (3):1–60, 2008.

Dominic Duggan. Type-based hot swapping of running modules. In *Proceedings of the International Conference on Functional Programming, ICFP '01, Florence, Italy*, pages 62–73. ACM Press, 2001.

Clarence Ellis and Karim Keddara. A workflow change is a workflow. In Wil van der Aalst, editor, *Business Process Management*, volume 1806 of *Lecture Notes in Computer Science*, pages 201–217. Springer-Verlag, 2000.

Clarence Ellis, Karim Keddara, and Grzegorz Rozenberg. Dynamic change within workflow systems. In *Proceedings of the Conference on Organizational Computing Systems, COOCS '95, Milpitas, CA, USA*, pages 10–21. ACM Press, 1995.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

Stephen Gilmore, Dilsun Kirli, and Christopher Walton. Dynamic ML without dynamic types. Technical Report ECS-LFCS-97-378, University of Edinburgh, 1997.

Michael Hicks. *Dynamic software updating*. PhD thesis, University of Pennsylvania, 2001.

Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Aksit and Satoshi Matsuoka, editors, *Proceedings of the European Conference on Object-Oriented Programming, ECOOP '97, Jyväskylä, Finland*, pages 220–242. Springer-Verlag, 1997.

Pieter Koopman, Rinus Plasmeijer, and Peter Achten. An executable and testable semantics for iTasks. In Sven-Bodo Scholz, editor, *Revised Selected Papers of the International Symposium on the Implementation and Application of Functional Languages, IFL '08, Hertfordshire, UK*, volume 5836 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009.

Thomas van Noort, Peter Achten, and Rinus Plasmeijer. A typical synergy - Dynamic types and generalised algebraic datatypes. In Marco Morazán and Sven-Bodo Scholz, editors, *Revised Selected Papers of the International Symposium on the Implementation and Application of Functional Languages, IFL '09, South Orange, NJ, USA*, volume 6041 of *Lecture Notes in Computer Science*, pages 179–197. Springer-Verlag, 2010a.

Thomas van Noort, Peter Achten, and Rinus Plasmeijer. Ad-hoc polymorphism and dynamic typing in a statically typed functional language. In Bruno Oliveira and Marcin Zalewski, editors, *Proceedings of the Workshop on Generic Programming, WGP '10, Baltimore, MD, USA*, pages 73–84. ACM Press, 2010b.

Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for GADTs. In Julia Lawall, editor, *Proceedings of the International Conference on Functional Programming, ICFP '06, Portland, OR, USA*, pages 50–61. ACM Press, 2006.

Marco Pil. Dynamic types and type dependent functions. In Kevin Hammond, Tony Davie, and Chris Clack, editors, *Proceedings of the International Workshop on the Implementation of Functional Languages, IFL '98, London, UK*, volume 1595 of *Lecture Notes in Computer Science*, pages 169–185. Springer-Verlag, 1999.

Rinus Plasmeijer, Peter Achten, and Pieter Koopman. iTasks: executable specifications of interactive work flow systems for the web. In Ralf Hinze and Norman Ramsey, editors, *Proceedings of the International Conference on Functional Programming, ICFP '07*, pages 141–152, Freiburg, Germany, 2007. ACM Press.

Manfred Reichert and Peter Dadam. ADEPT$_{\text{flex}}$ - Supporting dynamic changes of workflows without losing control. *Journal of Intelligent Information Systems*, 10(2):93–129, 1998.

Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: safe and predictable dynamic software updating. *ACM Transactions on Programming Languages and Systems*, 29(4), 2007.

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

Christopher Walton, Dilsun Kirli, and Stephen Gilmore. An abstract machine for module replacement. In Stephan Diehl and Peter Sestoft, editors, *Proceedings of the Workshop on Principles of Abstract Machines, PAM '98, Pisa, Italy*, 1998.

Barbara Weber, Manfred Reichert, and Stefanie Rinderle-Ma. Change patterns and change support features - Enhancing flexibility in process-aware information systems. *Data and Knowledge Engineering*, 66(3): 438–466, 2008.