An Executable and Testable Semantics for iTasks

Pieter Koopman, Rinus Plasmeijer, and Peter Achten

Nijmegen Institute for Computing and Information Sciences,
Radboud University Nijmegen, The Netherlands
{pieter, rinus, p.achten}@cs.ru.nl

Abstract. The iTask system is an easy to use combinator library for specifying dynamic data dependent workflows in a very flexible way. The specified workflows are executed as a multi-user web-application. The implementation of the iTask system is fairly complicated. Hence we cannot use it for reasoning about the semantics of workflows in the iTask system. In this paper we define an executable semantics that specifies how workflows react on events generated by the workers executing them. The semantics is used to explain iTasks and to reason about iTasks. Based on this semantics we define a mathematical notion of equivalence of tasks and show how this equivalence for tasks can be approximated automatically. Advantages of this executable semantics are: it is easy to validate the semantics by interactive simulation; properties of the semantics can be tested by our model-based test system G∀st. G∀st can test a large number of properties within seconds. These tests appeared to be a good indication about the consistency of the specified semantics and equivalence relation for tasks. The automatic testing of properties was very helpful in the development of the semantics. The contribution of this paper is a semantics for iTasks as well as the method used to construct this operational semantics.

1 Introduction

The iTask system [10] is an experimental toolkit to specify data dependent dynamic workflows in a flexible and concise way by a set of combinators. The iTask system supports workers executing the specified tasks by a web-based interface. Typical elementary user tasks in this system are filling in forms and pressing buttons to make choices. The elementary tasks are implemented on top of the iData system [9]. Based on an input the iTask system determines the new task that has to be done and updates the interface in the browser. Arbitrary complex tasks are created by combining (elementary) tasks. The real power of data dependent tasks is provided by the monadic bind operator that contains a *function* to generate the next task based on the value produced by the previous task.

The iTask implementation executes the tasks, but has to cope with many other things at the same time: e.g. i/o to files and database, generation of the multi-user web interface, client/server evaluation of tasks, and exception handling. The iTask system uses generic programming to derive interfaces to files, databases and web-browsers for data types. The combination of these things

makes the implementation of iTasks much too complicated to grasp the semantics. To overcome these problems we develop a high level operational semantics for iTasks in this paper. This semantics is used to explain the behavior of the iTask system, and to reason about the desired behavior of the system. In the future we will use this semantics as model to test the real iTask implementation with our model-based test tool G∀st. A prerequisite for model-based testing is an accurate model of the desired behavior. Making a model with the desired properties is not easy. Such a model is developed, validated, and its properties are tested in this paper.

In this paper we provide a basic rewrite semantics for iTasks as well as a number of useful notions to reason about tasks, such as *needed events* and *equivalence* of tasks. The semantics of many other workflow systems is based on Petri-nets [11], actor-oriented directed graphs (including some simple higher order constructs) [7], or abstract state machines (ASM) [6]. Neither of these alternatives is capable to express the flexibility covered by the dynamic generation of tasks of the monadic bind operation of the iTask system. As usual we omit many details in the semantics to express the meaning of the basic iTask combinators as clearly as possible. The semantics is expressed in the functional programming language Clean instead of the more common Scott Brackets, denotational semantics, or horizontal bar style, structural operational semantics a la Plotkin. The close correspondence between semantics and functional programs goes back at least to [8]. Expressing the operational semantics in a FPL is as concise as in Scott Brackets style. Using a functional programming language as carrier of the specification of the semantics has a number of advantages: **1)** the type system performs basic consistency checks on the semantics; **2)** the semantics is executable; **3)** using the iTask system it is easy to validate the semantics by interactive simulation; **4)** using the model-based test tool G∀st [5] it is possible to express properties about the semantics and equivalence of task concisely, and to test these properties fully automatically. Although the semantics is executable, it is not an iTask system itself. The semantics is a model of the real system, it lacks for instance a frontend (user interface) as well as a backend (e.g. interface to a database).

Especially the ability to express properties of the specified semantics and to test them automatically appears to be extremely convenient in the development of the semantics and associated notions described in this paper. An alternative, more traditional, approach would be to define a semantics in a common mathematical style, state properties in logic, and formally prove these properties. It would be wise to use a proof assistant like COQ [13] or SPARKLE [3] in proving the properties, this would require a transformation of the semantics to the language of the proof assistant. In the past we have used this approach for the iData system [1]. In such a mathematically based approach it is much harder to experiment with different formulations of the semantics and to get the system consistent after a change. Proving a property of a new version of the semantics typical takes some days of human effort where testing the same property is done in seconds by a computer. When we have a final version of the semantics

obtained in this way, we can decide to prove (some of) the tested properties in order to obtain absolute confidence in their correctness.

In section 2 we show how we model iTasks and the effect of applying an input to a task. In this section we also define useful notions about subtasks, such as when they are enabled or needed. In section 3 we define the equivalence of tasks and how the equivalence of tasks can be determined in two different ways. Some important properties of the semantics of iTasks are given in section 4, we also show how these properties can be tested fully automatically. Testing properties of the semantics increases the confidence that we have specified the semantics of iTasks right. In the future we will use this semantics for model-based testing of the real iTask implementation, this will increase the confidence that the system obeys the semantics. Finally there is a discussion.

2 A semantics for iTasks

In the original iTask system a task is a state transformer of the strict and unique Task State `TSt`. The required uniqueness of the task state (to guarantee single threaded use of the state in a pure functional language) is in Clean indicated by the type annotation `*`. The type parameter `a` indicates the type of the result. This result is returned by the task when it is completely finished.

```
:: Task a := *TSt → *(a,*TSt)    // an iTask is state transition of type TSt
```

Hence, a `Task` of type `a` is a function that takes a unique task state `TSt` as argument and produces a unique tuple with a value of type `a` and a new unique task state. In this paper we consider only one basic task: the edit task.

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

The function `editTask` takes a string and a value of type `a` as arguments and produces a task `a` under the context restriction that the type `a` is in the type class `iData`. The class `iData` is used to create a web based editor for values of this type. Here we assume that the desired instances are defined.

The `editTask` function creates a task editor to modify a value of the given type, and adds a button with the given name to finish the task. A user can change the value as often as she wants. The task is not finished until the button is pressed. There are predefined editors for all basic data types. For other data types an editor can be derived using Clean's generic programming mechanism [2], or a tailor-made editor can be defined for that type.

In this paper we focus on the following basic iTask combinators to compose tasks.

```
return          :: a                    → Task a      | iData a
(>>=) infixl 1  :: (Task a) (a→Task b)  → Task b      | iData b
(-||-) infixr 3 :: (Task a) (Task a)    → Task a      | iData a
(-&&-) infixr 4 :: (Task a) (Task b)    → Task (a,b)  | iData a & iData b
```

The combinators `return` and `>>=` are the usual monadic *return* and *bind*. The return combinator transforms a value in a task yielding that value immediately.

The bind combinator is used to indicate a sequence of tasks. The expression `t >>= u` indicates that first task `t` must be done completely. When this is done, its result is given to `u` in order to create a new task that is executed subsequently.

The expression `t -||- u` indicates that both iTasks can be executed in *any* order and *interleaved*, the combined task is completed *as soon as* any subtask is done. The result is the result of the task that completes first, the other task is removed from the system. The expression `t -&&- u` states that both iTasks must be done in any order (interleaved), the combined task is completed when *both* tasks are done. The result is a tuple containing the results of both tasks.

All these combinators are higher order functions manipulating the complex task state `TSt`. This higher order function based approach is excellent for constructing such a library in a flexible and type safe way. However, if we want to construct a program with which we can reason about iTasks, higher order functions are rather inconvenient. In a functional programming language like Haskell or Clean it is not possible to inspect which function is given as argument to a higher order function. The only thing we can do with such a function given as argument is applying it to arguments. In a programming context this is exactly what one wants to do with such a function. In order to specify the semantics of the various iTask combinators however, we need to know which operator we are currently dealing with. This implies that we need to replace the higher order functions by a *representation* that can be handled instead. We replace the higher order functions and the task state `TSt` by the algebraic data type `ITask`. We use infix constructors for the or-combinator, `.||.`, and the and-combinator, `.&&.`, in order to make the representation similar to the corresponding infix combinators `-||-` and `-&&-` from the original iTask library.

```
:: ITask
    = EditTask      ID      String  BVal            // an editor
    | .||. infixr 3 ITask   ITask                   // OR-combinator
    | .&&. infixr 4 ITask   ITask                   // AND-combinator
    | Bind          ID      ITask   (Val→ITask)     // sequencing-combinator
    | Return        Val                             // return the value

:: Val  = Pair Val Val  | BVal BVal
:: BVal = String String | Int Int   | VOID
```

Instances of this type `ITask` are called *task trees*. Without loss of generality we assume here that all editors return a value of a basic type (`BVal`). In the real iTasksystem editors can be used with every (user defined) data type. Using only these basic values in the semantics makes it easier to construct a type preserver simulator (see section 5). Since the right-hand side of the sequencing operator `Bind` is a normal function, this model has here the same rich expressibility as the real iTask system.

In order to write `ITask`s conveniently we introduce two abbreviations. For the monadic `Bind` operator we define an infix version. This operator takes a task and a function producing a new task as arguments and adds a default `id` to the `Bind` constructor.

```
(⇛) infixl 1 :: ITask (Val→ITask) → ITask
(⇛) t f = Bind id1 t f
```

For convenience we introduce also the notion of a button task. It executes the given iTask after the button with the given label is pressed. A button task is composed of a `VOID` editor and a `Bind` operator ignoring the result of this editor.

```
ButtonTask i s t = EditTask i s VOID ⇛ λ_ → t
```

Any executable form of iTasks will show only the button of a `VOID` editor. Since the `Type` of the edited value must be preserved, it cannot be changed.

2.1 Task Identification

The task to be executed is composed of elementary subtasks. These subtasks can be changed by events in the generated web-interface, like entering a value in a text-box or pushing a button. In order to link these events to the correct subtask we need an identification mechanism for subtasks. We use an automatic system for the identification of subtasks. Neither the worker, nor the developer of task specification has to worry about these identifications. The fact that the iTask system is in principle a multi-user system implies that there are multiple views on the task. Each worker can generate events independently of the other workers. The update of the task tree can generate new subtasks as well as remove subtasks of other workers. This implies that the id's of subtasks must be persistent and that newly generated subtasks cannot reuse old id's. For these reasons the numbering system has to be more advanced than just a numbering of the nodes. The semantics in this paper ignores the multi-user aspect of the semantics, but the numbering system is able to handle this (just as the real iTask system).

Tasks are identified by a list of integers. These task identifications are used similar to the sections in a book. On top level the tasks are assigned integer numbers starting at 0. In contrast to sections, the least significant numbers are on the head of the list rather than on the tail. The data type used to represent these task identifiers, `ID`, is just a list of integers.

```
:: ID = ID [Int]

next :: ID → ID
next (ID [a:x]) = ID [a+1:x]
```

Whenever a task is replaced by its successor the `id` is incremented by the function `next`. For every `id`, `i`, we have that `next i` $\neq$ `i`. In this way we distinguish inputs for a task from inputs to its successor. The function `splitID` generates a list of task identifiers for subtasks of a task with the given id. This function adds two numbers to the identifier, one number for the subtask and one number for the version of this subtask. If we would use the same number for both purposes, one application of the function `next` would incorrectly transform the identification of the current subtask to that of the next subtask.

```
splitID :: ID → [ID]
splitID (ID i) = [ID [0,j:i] \\ j ← [0..]]
```

These identifiers of subtasks are used to relate inputs to the subtasks they belong to. The function `nmbr` is used to assign fresh and unique identifiers to a task tree.

```
nmbr :: ID ITask → ITask
nmbr i (EditTask _ s v) = EditTask i s v
nmbr i (t .||. u)       = nmbr j t .||. nmbr k u where [j,k:_] = splitID i
nmbr i (t .&&. u)       = nmbr j t .&&. nmbr k u where [j,k:_] = splitID i
nmbr i (Bind _ t f)     = Bind k (nmbr j t) f    where [j,k:_] = splitID i
nmbr i t=:(Return _)    = t
```

By convention we start numbering with `id1 = ID [0]` in this paper.

2.2 Events

The inputs for a task are called *events*. This implies that the values of input devices are not considered as values that change in time, as in FRP (Functional Reactive Programming). Instead changing the value of an input device generates an event that is passed as an argument to the event handling function. This function will generate a new state and a new user interface.

An event is either altering the current value of an editor task or pressing the button of such an editor. At every stage of running an iTask application, several editor tasks can be available. Hence many inputs are possible. Each event contains the `id` of the task to which it belongs as well as additional information about the event, the `EventKind`.

```
:: Event     = Event ID EventKind | Refresh
:: EventKind = EE BVal | BE
```

The event kind `EE` (*E*ditor *E*vent) indicates a new basic value for an editor. A *B*utton *E*vent `BE` signals pressing the button in an editor indicating that the user finished editing.

Apart from these events there is a `Refresh` event. In the actual system it is generated by each refresh of the user-interface. In the real iTask system this event has two effects: 1) the task is normalized; and 2) an interface corresponding to the normalized task is generated. In the semantics we only care about the normalization effect. *Normalization* of a task is done by applying the `Refresh` event to the task. Although this event is ignored by all elementary subtasks it has effects on subtasks that can be rewritten without user events. For instance, the task `editTask "ok" 1 -||- return 5` is replaced by `return 5`. Similarly the task `return 7 >>= editTask "ok"` is replaced by `editTask "ok" 7` We elaborate on normalization in the next section.

2.3 Rewriting Tasks given an Event

In this section we define a rewrite semantics for iTasks by defining how a task tree changes if we apply an event to the task. Because we want an executable

```
instance @. ITask Event                                                    1
where                                                                      2
  (@.) (EditTask i n v) (Event j (EE w)) | i=:j = EditTask (next i) n w    3
  (@.) (EditTask i n v) (Event j BE)     | i=:j = Return (BVal v)          4
  (@.) (t .||. u) e  = case t @. e of                                      5
                         t=:(Return _) = t                                 6
                         t = case u @. e of                                7
                               u=:(Return _)  = u                          8
                               u              = t .||. u                   9
  (@.) (t .&&. u) e  = case (t @. e, u @. e) of                            10
                         (Return v, Return w) = Return (Pair v w)          11
                         (t, u)               = t .&&. u                   12
  (@.) (Bind i t f) e = case t @. e of                                     13
                         Return v = normalize i (f v)                      14
                         t        = Bind i t f                             15
  (@.) t e = t                                                             16
```

Fig. 1. The basic semantics of iTasks.

semantics rewriting is defined by an operator `@.`, pronounced as *apply*. We define a class for `@.` in order to be able to overload it, for instance with the application of a list of events to a task.

```
class (@.) infixl 9 a b :: a b → a
```

Given a task tree and an event, we can compute the new task tree representing the task after handling the current input. This is handled by the most important instance of the operator `@.` for `ITask` and `Event` listed in figure 1. It is assumed that the task is properly numbered and normalized, and that the edit events have the correct type for the editor.

This semantics shows that the `id`s play a dominant role in the rewriting of task trees. An event only has an effect on a task with the same `id`. Edit tasks can react on button events (line 4) as well as edit events (line 3). Line 14 shows why the `Bind` operator has an id. Events are never addressed to this operator, but the id is used to normalize (and hence number) the new subtask that is dynamically generated by `f v` if the left-hand side task is finished. All events that are not enabled are ignored (line 16). All other constructs pass the events to their subtasks and check if the root of the task tree can be rewritten after the reduction of the subtasks. The recursive call with `@. e` on line 13 can only have an effect when the task was not yet normalized, in all other situations applying the event has no effect.

A properly numbered task tree remains correctly numbered after reduction. Editors that receive a new value get a new number by applying the function `next` to the task identification number. The numbering scheme used guarantees that this number cannot occur in any other subtask. If the left left-hand task of the bind-operator is rewritten to a normal form a new task tree is generated by `f v`. The application of `normalize (next i)` to this tree guarantees that this tree

is well formed and properly numbered within the surrounding tree. This implies that applying an event repeatedly to a task has at most once an effect.

The handling of events for a task tree is somewhat similar to reduction of combinator systems or in the λ-calculus. An essential difference of such a reduction system with the task trees considered here is that all needed information is available inside a λ-expression. The evaluation of task trees needs the event as additional information.

Event sequences are handled by the following instance of the apply operator:

```
instance @. t [e] | @. t e where (@.) t es = foldl (@.) t es
```

Normalization A task `t` is *normalized* (or *well formed*) iff `t @. Refresh` = `t`. The idea is that all reductions in the task tree that can be done without a new input should have been done. In addition we require that each task tree considered is properly numbered (using the algorithm `nmbr` in section 2.1). In the definition of the operator `@.` we assume that the task tree given as argument is already normalized. Each task can be normalized and properly numbered by applying the function `normalize1` to that task.

```
normalize :: ID ITask → ITask
normalize i t = nmbr i (t @. Refresh)

normalize1 :: ITask → ITask
normalize1 t = normalize id1 t
```

Enabled Subtasks All editor tasks that are currently part of the task tree are *enabled*, which implies that they can be rewritten if the right events are supplied. The subtasks that are generated by the function on the right-hand side of a `Bind` construct are **not** enabled, even if we can predict exactly what subtasks will be generated. Events accepted by the enabled subtasks are called *enabled events*, this is the set of events that have an effect on the task when it is applied to such an event. Consider the following tasks:

```
t1 = EditTask id1 "b" (Int 1) .&&. EditTask id2 "c" (Int 2)
t2 = EditTask id1 "b" (Int 1) .||. EditTask id2 "c" (Int 2)
t3 = ButtonTask id1 "b" (EditTask id2 "c" (Int 3))
t4 = ButtonTask id1 "b" t4
t5 = EditTask id1 "b" (Int 5) ⇛ λv.ButtonTask id2 "c" (Return (Pair v v))
t6 = EditTask id1 "b" (Int 6) ⇛ λv.t6
t7 v p = EditTask id1 "ok" v  ⇛ λr=:(BVal w).if (p w) (Return r) (t7 w p)
```

In `t1` and `t2` all integer and button events with identifier `id1` and `id2` are enabled. In `t3` and `t4` only the event `Event id1 BE` is enabled. In `t5`, `t6` and `t7` all integer and button events with identifier `id1` are enabled. All other events can only be processed after the button event for the task with `id1` on the left-hand side of the bind operator.

Task `t4` rewrites to itself after a button event. In `t6` the same effect is reached by a bind operator. The automatic numbering system guarantees that the tasks

obtain another `id` after applying the enabled button events. Task `t7` is parameterized with a basic value and a predicate on such a value, and terminates only when the worker enters a value satisfying the predicate. This simple example shows that the bind operator is more powerful than just sequencing fixed tasks. In fact any function of type `Val→ITask` can be used there.

Normal Form A task is in *normal form* if it has the form `Return v` for some value `v`. A task in normal form is not changed by applying any event. The function `isNF :: ITask → Bool` checks if a task is in normal form. In general a task tree does not have a unique normal form. The normal form obtained depends on the events applied to that task. For task `t2` above the normal form of `t2 @. Event id1 BE` is `Return (BVal (Int 1))` while `t2 @. Event id2 BE` is `Return (BVal (Int 2))`. The recursive tasks `t4` and `t6` do not have a normal form at all.

Needed Events An event is *needed* in task `t` if the subtask to which the event belongs is enabled and the top node of the task tree `t` cannot be rewritten without that event.

In task `t1` above the events `Event id1 BE` and `Event id2 BE` are needed. Task `t2` has no needed event. This task can evaluate to a normal form by applying either `Event id1 BE` or `Event id2 BE`. As soon as one of these events is applied, the other task disappears. In `t3` only `Event id1 BE` is needed, the event `Event id2 BE` is not enabled. Similarly, in `t4`, `t5` and `t6` (only) the event `Event id1 BE` is needed.

For an edit-task the button-event is needed. Any number of edit-events can be applied to an edit-task, but they are not needed. For the task `t1 .&&. t2` the needed events is the sum of the needed events of `t1` and the needed events of `t2`. For a monadic bind the only needed events are the needed events of the left hand task. The needed events of a task `t` are obtained by `collectNeeded`. To ensure that needed events are collected in a normalized task we apply `normalize1` before scanning the task tree. In the actual iTask system the task is normalized by the initial refresh event and needs no new normalization ever after. In the task `t1 .||. t2` non of the events is needed, the task can is finished as soon as the task `t1` or the task `t2` is finished. Normalization is only include here to ensure that the task is normalized in every application of this function.

```
collectNeeded :: ITask → [Event]
collectNeeded t = col (normalize1 t)
where
  col (EditTask id n v) = [Event id BE]
  col (t1 .&&. t2)      = col t1 ++ col t2
  col (Bind id t f)     = col t              // no events from f
  col _                 = []                 // Return and the OR-combinator
```

In exactly the same spirit `collectButtons` collects all enabled button events in a task tree, and `collect` yields all enabled button events plus the enabled edit events containing the current value of the editors. The list of events is needed for the simulation of the task discussed in section 5.

An event is *accepted* if it causes a rewrite in the task tree, i.e. the corresponding subtask is enabled. A sequence of events is accepted if each of the events causes a rewrite when the events are applied in the given order. This implies that an accepted sequence of events can contain events that are not needed, or even not enabled in the original tree. In task `t2` the button event with `id1` and `id2` are accepted, also the editor event `Event id1 (EE (Int 42))` is accepted. All these events are enabled, but neither of them is needed. The task `t5` accepts the sequence `[Event id1 BE, Event id2 BE]`. The second event is not enabled in `t5`, but applying `Event id1 BE` to `t5` enables it.

Value The *value* of a task is the value returned by the task if we repeatedly press the left most button in the task until it returns a value. This implies that the value of task `t1` is `Pair (Int 1) (Int 2)`, the value of `t2` is `Int 1` since buttons are pressed from left to right. The value of `t3` is `Int 3` and the value of `t5` is `Pair (Int 5) (Int 5)`. The value of `t4` and `t6` is undefined. Since a task cannot produce a value before all needed events are supplied, we can apply all needed events in one go (there is no need to do this from left to right).

For terminating tasks the value can be computed by inspection of the task tree, there is no need to do the actual rewrite steps as defined by the `@.` operator. For nonterminating tasks the value is undefined, these tasks will never return a value. The class `val` determines the value by inspection of the data structure.

```
class val a :: a → Val

instance val BVal where val v = BVal v
instance val Val  where val v = v
instance val ITask
where
    val (EditTask i n e)    = val e
    val (Return v)          = val v
    val (t .||. u)          = val t    // priority for the left subtask
    val (t .&&. u)          = Pair (val t) (val u)
    val (Bind i t f)        = val (f (val t))
```

The value produced is always equal to the value returned by the task if the user presses all needed buttons and the leftmost button if there is no needed button. The property `pVal` in section 4 states this and testing does not reveal any problems with this property.

The value of a task can change after applying an edit event. For instance the value of task `EditTask id1 "ok" (BVal (Int 2))` is `BVal (Int 2)`. After applying `Event id1 (BVal (Int 7))` to this task the value is changed to `BVal (Int 7)`.

Type Although all values that can be returned by a task are represented by the type `Val`, we occasionally want to distinguish several families of values within this type. This type is not the data type `Val` used in the representation of tasks, but the type that the corresponding tasks in the real iTask system would have. We assign the type *Int* to all values of the form `Int i`. All values of the form

`String s` have type *String*. If value `v` has type v and value `w` has type w then the value `Pair v w` has type $Pair\ v\ w$. The types allowed are:

$$Type = Int \mid String \mid VOID \mid Pair\ \ Type\ \ Type$$

To prevent the introduction of yet another data type we represent the types yielded by tasks in this paper as instance of `Val`. The type *Int* is represented by `Int` 0 and the type *String* is represented as `String ""`. We define a class `type` to determine types of tasks.

```
:: Type := Val
class type a :: a → Type
```

Instances of this class for `Val` and `ITask` are identical to the instances of `val` defined in section 2.3. Only the instance for `BVal` is slightly different:

```
instance type BVal
where
    type (Int i)    = BVal (Int 0)
    type (String s) = BVal (String "")
    type VOID       = BVal VOID
```

3 Equivalence of Tasks

Given the semantics of iTasks we can define equivalence of tasks. Informally we want to consider two tasks equivalent if they have the same semantics. Since we can apply infinitely many update events to each task that contains an editor we cannot determine equivalence by applying all possible input sequences. Moreover, tasks containing a bind operator also contain a function and the equivalence of functions is in general undecidable. iTasks are obviously Turing complete and hence equivalence is also for this reason known to be undecidable. It is even possible to use more general notions of equivalence, like tasks are equivalent if they can be used to do the same job. Hence, developing a useful notion of equivalence for tasks is nontrivial.

In this paper we will develop a rather strict notion of equivalence of tasks: tasks t and u are equivalent if they have an equal value after all possible sequences of events and at each intermediate state the same events are enabled. Since the identifications of events are invisible for the workers using the iTask system, we allow that the lists of events applied to t and u differ in the event identifications. The strings that label the buttons in t and u do not occur in the events, hence it is allowed that these labels are different for equivalent tasks.

First we introduce the notion of *simulation*. Informally a task u can simulate a task t if a worker can do everything with u that can be done with t. It is very well possible that a worker can do more with u than with t. The notation $t \preccurlyeq u$ denotes that u can simulate t. Technically we require that: **1)** for each sequence of accepted events of t there is a corresponding sequence of events accepted by u; **2)** the values of the tasks after applying these events is equal; and **3)** after

applying the events, all enabled events of t have a matching event in u. Two events are equivalent, $e_1 \cong e_2$, if they differ at most in their identification.

$$t \preccurlyeq u \equiv \forall\, i \in \text{accept}(t).\exists j \in \text{accept}(u).i \cong j \wedge \mathit{val}(t\,@.\,i) = \mathit{val}(u\,@.\,j) \\ \wedge \mathit{collect}(t\,@.\,i) \subseteq \mathit{collect}(u\,@.\,j)$$

The notion $t \preccurlyeq u$ is not symmetrical, it is very well possible that u can do much more than t. As an example we have that for all tasks t and u that are not in normal form $t \preccurlyeq t\,.||.\,u$, and $t \preccurlyeq u\,.||.\,t$. If one of the tasks is in normal form it has shape `Return v`, after normalization the task tree $u\,.||.\,t$ will have the value `Return v` too. Any task can simulate itself $t \preccurlyeq t$, and an edit task of any basic value `v` can simulate a button task that returns that value: `ButtonTask id1 "b" (Return (BVal v))` $\preccurlyeq$ `EditTask id2 "ok" v`. In general we have $t\,.||.\,t \not\preccurlyeq t$: for instance if t is an edit task, in $t\,.||.\,t$ we can put a new value in one of the editors and produce the original result by pressing the `ok` button in the other editor, the task t cannot simulate this. The third requirement in the definition above is included to ensure that $t\,.||.\,t \not\preccurlyeq t$ also holds for tasks with only one button `ButtonTask id1 "b1" (BVal (Int 36))`.

Two tasks t and u are considered to be *equivalent* iff t simulates u and u simulates t.

$$t \cong u \;\equiv\; t \preccurlyeq u \;\wedge\; u \preccurlyeq t$$

This notion of equivalence is weaker then the usual definition of bisimulation [12] since we do not require equality of events, but just equivalency. Two editors containing a different value are not equivalent. There exist infinitely many event sequences such that these editors produce the same value. But for the input sequence consisting only of the button event, they produce a different value.

Since each task can simulate itself ($t \preccurlyeq t$), any task is equivalent to itself: $t \cong t$. If t and u are tasks that are not in normal form we have $t\,.||.\,u \cong u\,.||.\,t$. Consider the following tasks:

```
u1 = ButtonTask id1 "b1" (Return (BVal (Int 1)))
u2 = EditTask id2 "b2" (Int 1)
u3 = EditTask id2 "b3" (Int 2)
u4 = EditTask id2 "b4" (String "Hi")
u5 = u1 .||. u2
u6 = u2 .||. u1
u7 = u2 .&&. u4
u8 = u4 .&&. u2
u9 = u2 ⇛ λv.Return (BVal (Int 1))
u10 = u2 ⇛ λx.u4 ⇛ λy.Return (Pair x y)
```

The trivial relations between these tasks are $\mathrm{u}_i \preccurlyeq \mathrm{u}_i$ and $\mathrm{u}_i \cong \mathrm{u}_i$ for all u_i. The nontrivial relations between these tasks are: `u1` $\preccurlyeq$ `u2`, `u1` $\preccurlyeq$ `u5`, `u1` $\preccurlyeq$ `u6`, `u1` $\preccurlyeq$ `u9`, `u2` $\preccurlyeq$ `u5`, `u2` $\preccurlyeq$ `u6`, `u5` $\preccurlyeq$ `u6`, `u6` $\preccurlyeq$ `u5`, `u10` $\preccurlyeq$ `u7`, `u10` $\preccurlyeq$ `u8`, and `u2` $\cong$ `u9`, `u5` $\cong$ `u6`. Note that `u7` $\ncong$ `u8` since the tasks yield another value.

Due to the presence of functions in the task expressions it is in general undecidable if one task simulates another or if they are equivalent. However, in many

situations we can decide these relations between tasks by inspection of the task trees that determine the behavior of the tasks.

3.1 Determining the Equivalence of Task Trees

The equivalence of tasks requires an equal result for all possible sequences of accepted events. Even for a simple integer edit task there are infinitely many sequences of events. This implies that checking equivalence of tasks by applying all possible sequences of events is in general impossible.

In this section we introduce two algorithms to approximate the equivalence of tasks. The first algorithm, section 3.2, is rather straightforward and uses only the enabled events of a task tree and the application of some of these events to approximate equivalence. The second algorithm, section 3.3 is somewhat more advanced and uses the structure of the task trees to determine equivalence whenever possible.

We will use a four valued logic as for the result:

```
:: Result = Proof | Pass | CE | Undef
```

The result `Proof` corresponds to `True` and indicates that the relation is known to hold. The result `CE` (for *C*ounter*E*xample) is equivalent to `False`, the relation does not hold. The result `Pass` indicates that functions are encountered during the scanning of the trees. For the values tried the properties holds. The property might hold for all other values, but it is also possible that there exists inputs to the tasks such that the property does not hold. The value `Undef` is used as result of an existential quantified property ($\exists\, w.P\ x$) where no proof is found in the given number of test cases; the value of this property is undefined [5]. This type `Result` is a subset of the possible test results handled by the test system G∀st. For these results we define disjunction ('or', $\vee$), conjunction ('and', $\wedge$), and negation ('not', $\neg$) with the usual binding power and associativity. In addition we define the type conversion from Boolean to results and the weakening of a result which turns `Proof` in `Pass` and leaves the other values unchanged.

```
class (∨) infixr 2 a b :: a b → Result   // a OR b
class (∧) infixr 3 a b :: a b → Result   // a AND b

instance ¬ Result                        // negation

toResult :: Bool → Result                // type conversion
toResult b = if b Proof CE

pass :: Result → Result                  // weakens result to at most Pass
pass r = r ∧ Pass
```

For $\vee$ and $\wedge$ we define instances for all combinations of `Bool` and `Result` as a straightforward extension of the corresponding operation on Booleans.

3.2 Determining Equivalence by Applying Events

In order to compare `ITasks` we first ensure that they are normalized and supply an integer argument to indicate the maximum number of reduction steps. The value of this argument `N` is usually not very critical. In our tests 100 and 1000 steps usually gives identical (and correct) results. The function `equivalent` first checks if the tasks are returning currently the same value. If both tasks need inputs we first check **1)** if the tasks have the same type, **2)** if the tasks currently offer the same number of buttons to the worker, **3)** if the tasks have the same number of needed buttons, and **4)** if the tasks offer equivalent editors. Whenever either of these conditions does not hold the tasks `t` and `u` cannot be equivalent. When these conditions hold we check equivalence recursively after applying events. If there are needed events we apply them all in one go, without these events the tasks cannot produce a normal form. If the tasks have no needed events we apply all combinations of button events and check if one of these combinations makes the tasks equivalent. We need to apply all combinations of events since all button events are equivalent. All needed events can be applied in one go since they are needed in order to reach a normal form and the order of applying needed events is always irrelevant. If there are edit tasks enabled, `length et>0`, in the task the result is at most `Pass`. This is achieved by applying the functions `pass` or `id`.

```
equivOper :: ITask ITask → Result
equivOper t u = equivalent N (normalize1 t) (normalize1 u)

equivalent :: Int ITask ITask → Result
equivalent n (Return v) (Return w) = v == w
equivalent n (Return v) _ = CE
equivalent n _ (Return w) = CE
equivalent n t u
 | n≤0
   = Pass
   = if (length et>0) pass id
     (type t == type u ∧ lbt == lbu  ∧ lnt == lnu ∧ sort et == sort eu
     ∧ if (lnt>0)
           (equivalent (n-lnt) (t @. nt) (u @. nu))
           (exists N [equivalent n (t @. i) (u @. j)\\(i,j)←diag2 bt bu]))
where
    bt  = collectButtons t; nt  = collectNeeded t
    bu  = collectButtons u; nu  = collectNeeded u
    et  = collectEdit t;    eu  = collectEdit u
    lnt = length nt; lnu = length nu; lbt = length bt; lbu = length bu
```

The function `exists` checks if one of the first `N` values is `Pass` or `Proof`.

```
exists :: Int [Result] → Result
exists n []     = CE
exists 0 l      = Undef
exists n [a:x] = a ∨ exists (n-1) x
```

In this approach we do not apply any edit events. It is easy to design examples of tasks where the current approximation yields `Pass`, but applying some edit events

reveals that the tasks are actually not equivalent (e.g. `t` = `EditTask id1 (BVal (Int 5))` and `t` ⇛ `Return (BVal (Int 5))`). We obtain a better approximation of the equivalence relation by including some edit events in the function `equivalent`. Due to space limitations and to keep the presentation as simple as possible we have not done this here.

3.3 Determining Equivalence of Tasks by Comparing Task Trees

Since the shape of the task tree determines the behavior of the task corresponding to that task tree, it is tempting to try to determine properties like $t \preccurlyeq u$ and $t \cong u$ by comparing the shapes of the trees for u and t. For most constructs in the trees this works very well. For instance it is much easier to look at the structure of the tasks `EditTask id1 "ok" (BVal (Int 5))` and `EditTask id2 "done" (BVal (Int 5))` to see that they are equivalent, than approximating equivalence of these tasks by applying events to these tasks and comparing the returned values. In this section we use the comparison of task trees to determine equivalence of tasks. The function `eqStruct` implements this algorithm.

There are a number of constructions that allow different task trees for equivalent tasks. These constructs require special attention in the structural comparison of task trees:

1. The tasks `ButtonTask id1 "b" (Return v) .&&. Return w` and `ButtonTask id1 "b" (Return (Pair v w))` are equivalent for all basic values `v` and `w`. This kind of equivalent tasks with a different task tree can only occur if one of the branches of `.&&.` is in normal form and the other is not. On lines 9, 16 and 17 of the function `eqStruct` there are special cases handling this. The problem is handled by switching to a comparison by applying events, very similar to the `equivalent` algorithm in the previous section. The function `equ` takes care of applying events and further comparison.
2. The choice operator `.||.` should be commutative, (`t.||.u` ≃ `u.||.t`), and associative (`(t.||.u).||.v` ≃ `t.||.(u.||.v)`). In order to guarantee this, `eqStruct` collects all adjacent or-tasks in a list and checks if there is a unique mapping between the elements of those list such that the corresponding subtasks are equivalent (using `eqStruct` recursively). The implementation of the auxiliary functions is straightforward.
3. The `Bind` construct contains real functions, hence there are many ways to construct equivalent tasks with a different structure. For instance, we have that any task `t` is equivalent to the task `t` ⇛ `Return`, or slightly more advanced: `s.&&.t` is equivalent `(t .&&. s)` ⇛ λ`(Pair x y)`→`Return (Pair y x)` for all tasks `s` and `t`.
   The function `eqStruct` checks if the left-hand sides and the obtained right-hand sides of two bind operators are equivalent. If they are not equivalent the tasks are checked for equivalence by applying inputs, see line 13-15.

The `eqStruct` algorithm expects normalized task trees. The operator ≃ takes care of this normalisation.

```
class (≃) infix 4 a :: a a → Result      // is arg1 equivalent to arg2?

instance ≃ ITask where (≃) t u = eqStruc N (normalize1 t) (normalize1 u)
```

If the structures are not equal, but the task might be event equal we switch to applying inputs using the function `equ`. This function is very similar to the function `equivalent` in the previous section. The main difference is that the function `equ` always switches to `eqStruct` instead of using a recursive call. If a structural comparison is not possible after applying an event, the function `eqStruct` will switch to `equ` again.

```
eqStruc :: Int ITask ITask → Result                                          1
eqStruc 0 t u = Pass                                                         2
eqStruc n (Return v)       (Return w)       = v ≃ w                          3
eqStruc n (Return v)       _                = CE                             4
eqStruc n _                (Return w)       = CE                             5
eqStruc n (EditTask _ _ e) (EditTask _ _ f) = e≃f                            6
eqStruc n s=:(a .&&. b)    t=:(x .&&. y)                                     7
 = eqStruc (n-1) a x ∧ eqStruc (n-1) b y ∀                                   8
   ((inNF a || inNF b || inNF x || inNF y) ∧ equ n s t)                       9
eqStruc n s=:(a .||. b)    t=:(x .||. y)                                     10
 = eqORn n (collectOR s) (collectOR t)                                       11
eqStruc n s=:(Bind i a f) t=:(Bind j b g)                                    12
 = eqStruc (n-1) a b ∧ eqStruc (n-2) (f (val a)) (g (val b)) ∀ equ n s t     13
eqStruc n s=:(Bind _ _ _) t              = equ n s t                         14
eqStruc n s               t=:(Bind _ _ _) = equ n s t                        15
eqStruc n s=:(a .&&. b)   t              = (inNF a||inNF b) ∧ equ n s t      16
eqStruc n s               t=:(x .&&. y)  = (inNF x||inNF y) ∧ equ n s t      17
eqStruc n s               t              = CE                                18
```

This uses instances of ≃ for basic values (`BVal`) and values (`Val`). For these instances no approximations are needed. The line 10 and 11 implements the commutativity of the operator `.||.`: `collectOR` produces a list of all subtasks glued together with this operator, and `eqORn` determines if these lists of subtasks are equivalent in some permutation. The definitions are a direct generalization of the ordinary equality =.

A similar approach can be used to approximate the simulation relation ≼.

Property `pEquiv` in the next section states that both notions of equivalence yield equivalent results, even if we include edit events. Executing the associated tests indicate no problems with this property. This test result increases the confidence in the correct implementation of the operator ≃. Since ≃ uses the structure of the tasks whenever possible, it is more efficient than `equivOper` that applies events until the tasks are in normal form. The efficiency gain is completely determined by the size and contents of the task tree, but can be significant. It is easy to construct examples with an efficiency gain of one order of magnitude or more.

4 Testing Properties of iTasks

Above we mentioned a number of properties of iTasks and their equivalency like $\forall\ s, t \in$ iTask `.s.||.t≃t.||.s`. Although we designed the system such that these properties should hold, it is good to verify that the properties do hold indeed. Especially during the development of the system many versions are created in order to find a concise formulation of the semantics and an effective check for equivalence.

Creating formal proofs for all properties for all those versions of the semantics during its development is completely infeasible. Assuming that all well-typed versions of the semantics are correct is much to optimistic. We used the automatic test system G∀st to check the semantic functions presented here with a set of desirable properties. For instance the above property can be stated in G∀st as:

```
pOr :: GITask GITask → Property
pOr x y = normalize1 (t.||.u) ≃ normalize1 (u.||.t)
where t = toITask x; u = toITaskT (type t) y
```

The arguments of such a property are treated by the test system as universal quantified variables over the given types. The test system generates test values of these types using the generic class `ggen`. Since some `ITask` constructs contain a function, we use an additional data type, `GITask`, to generate the desired instances. We follow exactly the approach as outlined in [4]. The type `GITask` contains cases corresponding to the constructors in `ITask`, for button tasks, for tasks of the form `t ⇛ Return`, and for some simple recursive terminating tasks. For `pOr` we need to make sure the tasks `t` and `u` have the same type since we combine them with an or-operator. The conversion by `toITask` from the additional type `GITasks` used for the generation to `ITasks` takes care of that.

After executing 23 tests G∀st produces the first counterexample that shows that this property does not hold for `t = Return (BVal (Int 0))` and `u = Return (Pair (BVal (Int 0)) (BVal (Int 0)))`. Using the semantics from figure 1 it is clear that G∀st is right, our property is too general. A correct property imposes the condition that `t` and `u` are not in normal form:

```
pOr2 x y = notNF [t,u] ==> normalize1 (t.||.u) ≃ normalize1 (u.||.t)
where t = toITask x; u = toITaskT (type t) y
```

In the same way we can show that `t.||.t ≱ t` for tasks that are not in normal form (`p2`) and test the associativity of the `.||.` or operator (`p3`).

```
p2 :: GITask GITask → Property
p2 x y = notNF [s,t] ==> (s.||.t)≱t
where s = toITask x; t = toITaskT (type s) y

p3 :: GITask GITask GITask → Property
p3 x y z = (s .||. (t .||. u))≃((s .||. t) .||. u)
where s = toITask x; t = toITaskT (type s) y; u = toITaskT (type s) z
```

In total we have defined over 70 properties to test the consistency of the definitions given in this paper. We list some representative properties here. The first

property states that needed events can be applied in any order. Since there are no type restrictions on the type `t` we can quantify over `ITask`s directly.

```
pNeeded :: ITask → Property
pNeeded t = (λj. t @. i ≃ t @. j) For perms i where i = collectNeeded t
```

In this test the fragment `For perms i` indicates an additional quantification over all `j` in `perms i`. The function `perms :: [x] → [[x]]` generates all permutations of the given list. In logic this property would have been written as $\forall t \in \mathsf{ITask}$, $\forall j \in \mathsf{perms}\ (\mathsf{collectNeeded}\ t).\ t\ @.\ (\mathsf{collectNeeded}\ t) \simeq t\ @.\ j$.

The next property states that both approximations of equivalence discussed in the previous section produce equivalent results.

```
pEquiv :: ITask ITask → Property
pEquiv t u = (equivOper t u) ≃ (t≃u)
```

The type of a task should be preserved under reduction. In the property `pType` also events that are not well typed will be tested. Since we assume that all events are well typed (the edit events have the same type as the edit task they belong to), it is better to use `pType2` where the events are derived from the task `t`.

```
pType :: ITask → Property
pType t = (λi.type t == type (t @. i)) For collect t

pType2 :: ITask → Property
pType2 t = pType t For collect t
```

The phrase `For collect t` indicates that for testing these properties the events are collect from the task tree rather then generated systematically by G∀st. However the tasks to be used in the test are generated systematically by G∀st.

The property `pVal` states that the value of a task obtained by the optimized function `val` is equal to the value of the task obtained by applying events obtained by `collectVal` until it returns a value. The function `collectVal` returns all needed events and the leftmost events if these are no needed events.

```
pVal :: ITask → Property
pVal t = val t == nf t
where
    nf (Return v) = v
    nf t = nf (t @. collectVal t)
```

The definitions presented in this paper pass all stated properties. On a normal laptop (Intel core2 Duo (using only one of the cores), 1.8 GHz) it takes about 7 seconds to check all defined properties with 1000 test cases for each property. This is orders of magnitude faster and more reliable then human inspection, which is on its turn much faster than a formal proof (even if it is supported by a state of the art tool). Most of these properties are very general properties, like the properties shown here. Some properties however check specific test cases that are known to be tricky, or revealed problems in the past. If there are problems with one of the properties, they are usually spotted within the first 50 test cases generated. It appears to be extremely hard to introduce flaws in the system that

are not revealed by executing these tests. For instance omitting one of the special cases in the function `eqStruct` is spotted quickly. Hence testing the consistency of the system in this way is an effective and efficient way to improve the confidence in its consistency.

5 Discussion

In this paper we give a rewrite semantics for iTasks. Such a semantics is necessary to reason about iTasks and their properties, it is also well suited to explain their behavior. In addition we defined useful notions about iTasks and stated properties related to them. The most important notion is the *equivalence* of tasks.

Usually the semantics of workflow systems is based on Petri nets, abstract state machines, or actor-oriented directed graphs. Since the iTask system allows arbitrary functions to generate the continuation in a sequence of tasks (the monadic bind operator), such an approach is not flexible enough. To cope with the rich possibilities of iTasks our semantics incorporates also a function to determine the continuation of the task after a `Bind` operator.

We use the functional programming language Clean as carrier for the semantical definitions. The tasks are represented by a data structure. The effect of supplying an input to such a task is given by an operator modifying the task tree. Since we have the tasks available as data structure we can easily extract information from the task, like the events needed or accepted by the task. A typical case of the operator `@.` (apply) that specifies the semantics is:

```
(@.) (EditTask i n e) (Event j BE) | i=j = Return (BVal e)
```

In the more traditional Scott Brackets style this alternative is written as:

$$\mathcal{A}\,[\![\, \mathit{EditTask}\ i\ n\ e \,]\!]\ (\mathit{Event}\ j\ \mathit{BE}) = \mathit{Return}\ (\mathit{BVal}\ e),\ \text{if}\ i = j$$

Our representation has the same level of abstraction and has as advantages that it can be checked by the type system and executed (and hence simulated and tested).

Having the task as a data structure it is easy to create an editor and simulator for tasks using the iTask library. Editing and simulating tasks is helpful to validate the semantics. Although simulating iTasks provides a way to interpret the given task, the executable semantics is not intended as an interpreter for iTasks. In an interpreter we would have focused on a nice interface and efficiency, the semantics focusses on clearness and simplicity.

Compared with the real iTask system there are a number of important simplifications in our `ITask` representation. **1)** Instead of arbitrary types, the `ITasks` can only yield elements of type `Val`. The type system of the host language is not able to prevent type errors within the `ITasks`. For instance it is possible to combine a task that yields an integer, `BVal (Int i)`, with a task yielding a string, `BVal (String s)`, using an `.||.` operator. In ordinary iTasks it is type technically not possible (and semantically not desirable) to combine tasks of type `Task Int` with `Task String` using a `-||-` operator. Probably GADTs would have helped us

to enforce this condition in our semantical representation. **2)** The application of a task to an event does not yield an HTML-page that can be used as GUI for the iTask system. In fact there is no notion at all of HTML output in the `ITask` system. **3)** There is no way to access files or databases in the `ITask` system. **4)** There is no notion of workers and assigning subtasks to them. **5)** There is no difference between client site and server site evaluation of tasks. **6)** There is only one workflow process which is implicit. In the real iTask system additional processes can be created dynamically. **7)** The exception handling from the real iTask system is missing in this semantics.

Adding these aspects would make the semantics more complicated. We have deliberately chosen to define a concise system that is as clear as possible.

Using the model-based test system it is possible to test the stated properties fully automatically. We maintain a collection of over 70 properties and test them with one push of a button. Within seconds we do known if the current version of the system obeys all properties stated. This is extremely useful during the development and changes of the system. Although the defined notions of equivalence are in general undecidable, the given approximation works very well in practice. Issues in the semantics or properties are found very quickly (usually within the first 100 test cases). We attempted to insert deliberately small errors in the semantics that are not detected by the automatic tests, but we failed miserably.

In the near future we want to test with G∀st if the real iTask system obeys the semantics given in this paper. In addition we want to extend the semantics in order to cover some of the important notions omitted in the current semantics, for instance task execution in a multi-user workflow system. When we are convinced about the quality and suitability of the extended system we plan to prove some of the tested properties. Although proving properties gives more confidence in the correctness, it is much more work then testing. Testing with a large number of properties has shown to be an extremely powerful way to reveal inconsistencies in the system.

Acknowledgement We thank the anonymous referees and the shephard for their suggestions to improve this paper.

References

1. P. Achten, M. van Eekelen, M. de Mol, and R. Plasmeijer. An Arrow based semantics for interactive applications. In M. Morazán, editor, *Preliminary Proceedings of the 8th Symposium on Trends in Functional Programming, TFP'07*, New York, NY, USA, 2-4, Apr. 2007.
2. A. Alimarine and R. Plasmeijer. A generic programming extension for Clean. In T. Arts and M. Mohnen, editors, *Selected Papers of the 13th International Symposium on the Implementation of Functional Languages, IFL'01*, volume 2312 of *LNCS*, pages 168–186. Springer-Verlag, Sept. 2002.
3. M. de Mol, M. van Eekelen, and R. Plasmeijer. The mathematical foundation of the proof assistant Sparkle. Technical Report ICIS-R07025, Institute for Computing

and Information Sciences, Radboud University Nijmegen, The Netherlands, Nov. 2007.
4. P. Koopman and R. Plasmeijer. Automatic testing of higher order functions. In N. Kobayashi, editor, *Proceedings of the 4th Asian Symposium on Programming Languages and Systems, APLAS'06*, volume 4279 of *LNCS*, pages 148–164, Sydney, Australia, 8-10, Nov. 2006. Springer-Verlag.
5. P. Koopman and R. Plasmeijer. *Fully automatic testing with functions as specifications*, volume 4164 of *LNCS*, pages 35–61. Springer-Verlag, Budapest, Hungary, 4-16, July 2006.
6. S.-Y. Lee, Y.-H. Lee, J.-G. Kim, and D. C. Lee. Workflow system modeling in the mobile healthcare B2B using semantic information. In *Proceedings of the 5th International Conference on Computational Science and its Applications, ICCSA'05*, pages 762–770. LNCS, 2005.
7. B. Ludäscher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific workflow management and the Kepler system. *Concurrency and Computation: Practice & Experience*, 18:2006, 2005.
8. H. R. Nielson and F. Nielson. *Semantics with applications: a formal introduction*. John Wiley & Sons, Inc., 1992.
9. R. Plasmeijer and P. Achten. iData for the world wide web - Programming interconnected web forms. In *Proceedings of the 8th International Symposium on Functional and Logic Programming, FLOPS'06*, volume 3945 of *LNCS*, Fuji Susone, Japan, 24-26, Apr. 2006.
10. R. Plasmeijer, P. Achten, and P. Koopman. iTasks: executable specifications of interactive work flow systems for the web. In *Proceedings of the 12th International Conference on Functional Programming, ICFP'07*, pages 141–152, Freiburg, Germany, 1-3, Oct. 2007. ACM Press.
11. N. Russell, A. ter Hofstede, and W. van der Aalst. newYAWL: specifying a workflow reference language using coloured Petri nets. In *Proceedings of the 8th Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, CPN'07*, 2007.
12. C. Stirling. The joys of bisimulation. In *Proceedings of the 23rd International Symposium on Mathematical Foundations of Computer Science, MFCS'98*, pages 142–151, London, UK, 1998. Springer-Verlag.
13. T. C. D. Team. *The Coq proof assistant reference manual (version 7.0)*, 1998. `http://pauillac.inria.fr/coq/doc/main.html`.