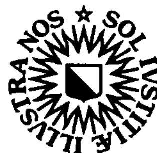
Modelling Office Processes with Functional Parsers

G. Florijn

UU-CS-1994-50
November 1994



Utrecht University

Department of Computer Science

Padualaan 14, P.O. Box 80.089,
3508 TB Utrecht, The Netherlands,
Tel. : ... + 31 - 30 - 531454

Modelling Office Processes with Functional Parsers

G. Florijn

Technical Report UU-CS-1994-50
November 1994

Department of Computer Science
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
The Netherlands

ISSN: 0924–3275

Modelling Office Processes with Functional Parsers

Gert Florijn

Utrecht University
Department of Computer Science
P.o. Box 80.089
3508 TB Utrecht, the Netherlands
E-mail: florijn@cs.ruu.nl

Abstract

Office work and software development have many things in common. For example, both are marked by complex information processes activities involving coordinated contributions by multiple workers and systems. Also, in both fields there is interest in process support systems: tools that assist people in performing, managing and (partially) automating these processes, using a description (or process program) of how the process is supposed to proceed.
The goal of the Ariadne project at Utrecht University is to develop a flexible process support system which can handle a diversity of processes, ranging from formalized, structured business processes to ad-hoc, evolving and semi-structured processes defined by individuals and groups. To cater for this diversity, we not only need a suitable process progamming language and a flexible environment, but also different styles of process enactment. For this latter point we consider it crucial to separate the operational issues surrounding process enactment from the conceptual modelling of the process as a space of legal events.
In this paper we explore the use of grammars and parsers for the conceptual modelling of processes. We represent process models as parsers created with parsing combinators and encoded in a functional language. The resulting formalism combines non-determinism, state and parallellism. We illustrate the use of the approach by several examples, including the ISPW software process model.

1. Introduction and background

Office work and software development have many things in common. For example, both involve several kinds of hybrid processes, complex information processing activities aimed at a particular goal and involving coordinated contributions by multiple workers and systems. It is not surprising therefore that both domains show a strong interest in process support technology. Many offices are considering the use of office procedure systems or workflow management tools [Florijn94a, Gulla94, Medina-Mora92, Leymann94] to support their routine business processes. Similarly, in the field of software engineering, researchers and practitioners are exploring process programming environments to support and automate software development processes [Kaiser93, Gruhn91, Pierce91, Zucker91, Yasumoto94].

In general, a process support system is a tool that helps people in defining, performing, managing and (partially) automating office processes. The system is generic in that it can support different kinds of processes at the same time. A particular type of process is described by a procedure (or process program) which – roughly speaking – is a program that describes how a particular kind of process should proceed. The procedure lists the activities that can occur, defines the possible flow of control among these activities and the data that is used and produced, constrains who should or can perform activities, etc. Actual processes that take place are executing instances of a procedure. The process support system maintains each process' state by keeping the data together in some electronic store and by tracking the activities that are performed. Using the procedure, the system can also play an active role: it can offer tasks to workers, remind them of their commitments, check results and send data from one worker to another.

Process support tools offer several (potential) advantages such as traceability of processes, consistency in handling and improved efficiency through faster communication. The fact that they can be parametrized by a reasonably high-level process description means that new applications can be developed quickly, and existing ones can be adapted easily.

1.1. Characteristics of hybrid processes

The use of the term "office process" above is somewhat of a simplification. Within an office – and similarly within software development organizations – there are various type of work, each with different characteristics [Hirschheim85]. The first category is formed by routine, formalized and structured business processes. These are events which occur frequently, which are well understood and for which a standard way of handling has been described in detail. Typically this involves administrative tasks like order processing, invoicing, insurance claim processing, etc.

Formalized process cover only (a small) part of office work. Another type of activity is ad-hoc, cooperative problem-solving performed in reaction to unforeseen events. When such unexpected situations occur within the context of a structured business process they may give rise to exceptions [Karbe90] and turn a routine process into an ad-hoc, non-formalized one. The ad-hoc nature does not mean that these activities are performed at random or unorganized. They involve contributions by different people, there is data that is produced and used, there are deadlines to be met, etc. In fact, there is a plan or procedure, but it is not defined in advance or in great detail. Rather it is created and adapted on the fly by the people involved, perhaps with the use of tools like planners [Croft88].

Routine events do not always involve a standard procedure. Often, the way in which a task is performed depends on who performs it or where it is performed (so-called situated action [Suchman87]). Decision making, planning and authoring are well-known examples. Situated action can also happen within the context of formal processes. For example, when a group of analysts have to reach a decision regarding a loan, this will be one task in a larger, formal process.

Finally, there are activities which are partly standardized. These are called semi-structured [Kaplan91] processes. Only some aspects of it are defined explicitly in a procedure; the others are left open or elaborated during the performance of a process. For example we can view an electronic conversation as a process in which the decomposition into sub-activities is very simple and the data is just a collection of message. In design argumentation models like IBIS [Conklin88], the activity-model is very simple, but the data structure is modelled more explicitly as a web of issues, positions and arguments. Finally, in a process encoding a versioning system, we have a reasonably detailed activity decomposition but do not place any constraints on the data objects that are versioned.

Although there do not seem to be many (empirical) studies into the nature of software development (see [Curtis88, Hofstede89, Verhoef93] for a few examples) there is no reason to believe why software development should be different from general office work in these issues. Of course, there are routine processes that are very similar to the business processes in offices. Every reasonably mature software development organization has procedures to deal with releases, or for handling changes to a product in a consistent way (see [Rose92, Harrison90] for some sample fragments). But obviously this does not mean that all software development activities can be standardized in great detail. Software development on a smaller scale – i.e. programming and design – is a creative activity. Developers have their own ways of working in dealing with particular problems [Clemm89] and these are difficult to standardize for use by others [Verhoef93]. This is one of the reasons why the initial proposal of process support for software processes [Osterweil87] was received with some scepsis [Lehman87, Curtis87]. Furthermore, even when a software process is planned in detail on a large abstraction level, the actual work will contain a lot of ad-hoc informal activities by teams.

1.2. Process support

The generality of the process metaphor suggests the possibility of a general process support system which can handle this diversity of processes. Such a system should handle interrelated processes for organizations, groups and individual users and allow structuring of processes/procedures along different dimensions (activity decomposition, control flow, data objects, role model and allocation constraints, etc.) and to different degrees of detail. Furthermore, it should offer generic support for procedure development and evolution (e.g. through analysis of process models, simulation or the gathering of run-time statistics) and process management and control (e.g. monitoring and tracking processes, notification of certain operational problems such as absence of employees, and analysis of process properties such as expected duration). And of course, it should provide means to handle exceptions and deal with ad-hoc processes. This includes the relaxation of constraints or the adaptation of a procedure within the context of a particular process (when permitted). Finally, from a technical perspective, the system should facilitate the use of existing tools for automated activities and the use of data stored in external systems, and should be able to operate in a distributed environment. Preferrably, the system should be able to operate in a loosely coupled environment where network connections are not permanently available. This facilitates the use of portable computers.

Existing process support tools however, do not meet these requirements. Workflow systems and office procedure systems lack the flexibility to deal adequately with exceptions and ad-hoc processes [Kreifelts91, Florijn94a]. In most systems procedures can only be defined by specialists or administrators and not by end-users. Furthermore, many systems do not offer the means to relax constraints or adapt a procedure within the context of a particular process which is needed for ad-hoc processes. Decomposition of a formal task into a lot of different activities by the members of the team also falls outside the scope of the workflow system. It only registers the fact that the higher level task has been started and completed.

More flexibility is offered in generic groupware systems that allow users to model (parts of) collaborative, semi-structured processes. The classical examples of this are the programmable

messaging systems through which users can organize and automate some of the work involved in handling electronic communication, such as filtering messages, automatic filing or forwarding, exchanging and manipulating forms [Hammainen90] or objects [Lai88], or even computational mail [Borenstein92]. In some cases a complete conversation model can be defined, instances of which are executed via e-mail communication with typed messages [Shepherd90] or as conversation objects operating on a shared hypertext data-store [Kaplan91, Kaplan92].

While these latter systems allow users to model semi-structured processes and provide more flexibility to handle ad-hoc processes and exceptions, there are two forms of parametrization that they do not provide. The first is parametrization of the style of enactment [Kaiser93, Heimbigner92]): the way in which a particular process "behaves" towards the user. Some processes (e.g. business processes) require active enactment where the system determines the tasks that are to be performed next and schedules their execution. Other kinds of processes (e.g. an electronic conversation) may require more passive enactment where users decide which task they want to work on next. In yet other cases the existence of a procedure may be implicit. Users may only see a visual representation of some data and modify it. In this case, the system should track the activities performed by the users and check whether these correspond to (enabled) actions in the procedure using some association between the data structure and the actions.

The second issue is the technical implementation of the process. While an electronic conversation is typically used for asynchronous interaction among people, a tool like gIBIS [Conklin88] provides facilities for real-time interaction. Updates to the web are immediately propagated to all other people working on it. However, we can also imagine the use of gIBIS in an asynchronous setting. While the process model remains the same in both cases, the technical implementation used to drive the process will be completely different.

1.3. Ariadne: general support for hybrid office processes

The goal of our research in the Ariadne project at Utrecht University [Florijn94b, Florijn94a] is to create an environment (called Ariadne) which supports the whole range of hybrid office processes identified earlier and meets the requirements sketched above. In addition to these, Ariadne should provide different styles of process enactment and technical implementation which can be applied to the same process, preferrably without changes to the procedure or process.

A primary objective in Ariadne is to bring support for office processes into the computing environment of end-users. The central ingredient therefore is the process, a context for (group) work aimed at a particular goal or task. Its basic component is a workspace or dossier, a dynamically extensible, tree-like data structure holding information relevant to the process. The workspace can hold arbitrary data-objects – including references to information in external systems – and can be refined dynamically. Our current work-hypothesis for dossiers is to represent them as ψ-terms or flexible records [Ait-Kaci86]. Events occurring in a process can ultimately be viewed as modifications or extensions to dossier, e.g. filling-out the fields in a form, linking a new version to a version graph, or adding messages to conversation.

Each process is governed by a procedure which defines the legal actions and action traces that can occur [Boerboom94, Florijn94c]. More specifically, the procedure constrains the actors (human user or tool) that can perform an action (e.g. by defining roles and binding-constraints), when an action can be performed (by describing dependencies and real time-constraints), and what the effects of an action on the dossier can be. Using object-oriented terms, the procedure defines the methods that can be performed on the "process object" and defines the protocol that coordinates when methods can be performed and by who [Nierstrasz93], see figure 1.

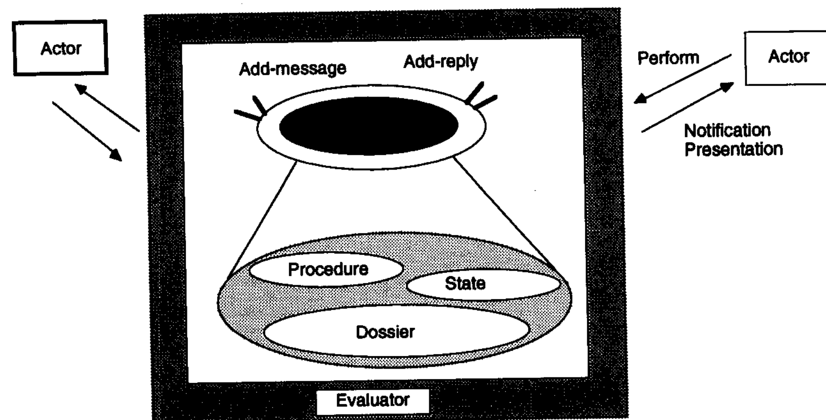


Figure 1: A process as an (active) object

Each process holds a local copy of the procedure that governs it together with an execution state and history. Processes thus are mostly self-contained objects (apart from references to external information), which makes the handling of distribution and migration simpler, even in loosely-coupled networks where connections are not permanently available. Users can – in principle – take a process along on a portable machine and bring it back later.

The local copy of a procedure can be modified whenever exceptions give rise to this and when permissions allow it. In order to do this, processes (consisting of a dossier, procedure and execution state) must have an editable representation. In fact, a complete process can be represented as a workspace data-structure which allows the introduction of reflective processes, i.e. processes that operate upon other processes. The typical example is a process in which the execution of another process is planned or adapted [Pemberton91, Bandinelli93].

Individual users can define their own procedures and create processes governed by existing procedures. When users are notified of the fact that they can perform an action in a particular process they can automate their way of handling of this task by specifying the initiation of a new process (governed by a different procedure) in their local interface agent [Asch92]. The system maintains the resulting relationships among processes.

1.4. Enactment styles

As mentioned above, an important goal of the Ariadne project is to provide different enactment styles for processes without having to change a procedure. This requires that we treat a procedure as a functional specification of the rules to which the work done by human and computer actors should conform. Phrased differently, the procedure should only describe legal actions and action events. The operational issues of how a procedure is to be enacted are "policies" describing the use of a procedure in a particular setting.

In our model they are handled by an evaluator which implements a particular enactment style. Typical examples of such operational issues involve the interaction of the process support system with the users: policies for finding a user to perform a role, notifying tem about a task they can perform, reminding them of deadlines, sending managers notifications of task completions, etc. The evaluator also processes events generated by users and tools, maps these to the actions defined in the procedure and checks whether the results conform to the constraints defined there. An eager evaluator will actively look for actors (users or tools) that can perform enabled actions and ask them to do the work. A more passive evaluator will just handle events and check for conformance. In a similar vein, the same procedure can be interpreted by evaluators that differ in technical matters. A procedure that coordinates updates on a document, for example, could be interpreted by an evaluator that handles asynchronous updates, but also by a synchronous one.

If a procedure is free of operational issues, we can see it as a definition of a set of legal action traces (a trace space). An action trace is a set of actions each of which corresponds to a elementary piece of work that can take place within the corresponding process. An action is characterized by properties like the actor that performs it, the location where it is performed, the data object it produced or the time at which it takes place. Furthermore, each action produces a data object, which will be stored in the workspace for the process. All the valid action traces, and the trace space as a whole, are subsets of the overall set of possible action sequences.

A process modelling language allows us to describe procedures. Clearly, it should allow for different enactment styles and allow incremental conformance checking. To express action traces [Boerboom94, Florijn94c], it should offer constructs to:

- define constraints on the result of an action, e.g. via a type definition for the data object
- constrain who can perform an action, e.g. by mentioning a specific actor, or by modelling binding constraints among actions (e.g. an actor who performed one action is not allowed to performed another action).
- define or constrain resources like the location (e.g. machine) where an action is performed
- define or constrain when work on actions can begin or end.

In addition, a practical process modelling language should:

- be simple to use and lead to procedure definitions that are easily understood.
- be extensible, so that new modelling constructs can be defined.
- have a formal basis, so that analysis of procedures and processes can be provided.
- make it easy to refine or adapt procedures, while offering coercion for existing processes.
- provide abstraction mechanisms to organize a procedure.
- offer means for instantiating constructs for a dynamically determined group of values.

1.5. Modelling processes as grammars

In the remainder of this paper we discuss the use of grammars encoded by functional parsers for describing procedures and checking conformance. A procedure is modelled by defining parsers that match individual actions and by glueing these together (using meta-constructs or parser combinators) into more complex parsers. The grammar thus is encoded in a program that accepts all legal action traces.

We assume in this paper that each (elementary or composite) action is associated directly with a unique part of the process' workspace. Elementary actions produce elementary values, while composite actions map to composite values according to standard rules. This means that the relationship between the procedure that coordinates the work and the workspace that contains the results is very close. One way to view a process in this model therefore is as a data-object that is created in a coordinated way.

Our main goals here are to explore the kinds of constructs that are needed, and to illustrate the capabilities of this approach. We describe several non-trivial processes, including the ISPW example [Heimbigner91]. To make this discussion concrete, we need a formalism to express grammars/parsers and the values they produce. We use Gofer – a functional programming language with lazy evaluation and strong typing – and express parsers through the use of parser-combinators [Hutton92]. More specifically, we model parsers using monads [Wadler90, Wadler92], because this allows us to abstract from the details of the implementation of the parsing, and supports the use of monad-comprehensions, a syntactic shorthand offered by Gofer that allows us to keep the examples simpler and more intuitive. In order to provide the necessary background, we start with a brief introduction of parser combinators. This also introduces some of the main characteristics of the Gofer language.

2. Parsing with combinators

A parser for a particular grammar can be seen as a function that takes a string of items and checks whether this string is a valid sentence, i.e. one defined by the grammar that the parser implements. We can associate semantic actions with the parser which are invoked during the parse process. These allow us to make more useful applications, e.g. to let the parser construct a parse-tree.

2.1. Parsers

In Gofer, we can define a parser as a function of the following type:

```
type Parser is a  =  is -> [( a, is )]
```

A parser takes a state denoted by `is` (typically, a list of tokens), and produces a list of tuples. Each tuple represents a succesful parse, that is, a result value of type a together with the adapted state (typically the remainder of the input that was not parsed). Note that this means that the grammar that is parsed can be non-deterministic. The parser produce all possible parses on a given input.

Parsers are constructed by combining elementary parsers. Two of the basic parsers are: `fail` which always fails, and okay which succeeds with a given value as its result. Another basic parser is `item`. It matches the next item of the input, and fails if no input is left.

```
fail     ::  Parser is a
fail xs   =  []

okay        :: a -> Parser is a
okay v xs    =  [ (v,xs) ]

item ::  Parser [a] a
item [] = fail []
item (i:is) = okay i is
```

Note that we assume here that the input state is modelled as a list of items of type a. The construct `(a:b)` denotes a list whose head element is a and whose tail is the list denoted by b.

2.2. Combinators

Elementary parsers are combined using combinators. The first one discussed here is `into` which performs a parser on the input and – if it was succesful – performs a second parser on the remaining input. The second parser gets the result of the first parser as an argument. `into` returns the result of the second parser. The second combinator is `orelse`. It applies both parsers on the current input and returns the combined results

```
infixr 6 `into`
infixr 4 `orelse`

into    :: Parser i a -> (a -> Parser i b) -> Parser i b
(p `into` q) xs = [(r,xs'') | (v,xs') <- p xs, (r, xs'') <- q v xs']

orelse :: Parser i a -> Parser i a -> Parser i a
(p1 `orelse` p2) xs   =  p1 xs ++ p2 xs
```

Note that `into` and `orelse` are used as infix operators, where `into` binds more strongly. As follows from the definition, `orelse` concatenates (the ++ operator) the results of attempting both alternatives, leading to the fact that a parser returns a list of sucesses. The definition of `into` uses list-comprehension. The expression `[r|v <- e1, r <- e2 v, p r]` denotes the list of r's obtained by iterating v over the values returned by `e1`, binding `r` to the result of applying v to `e2` and checking whether `p v` is true.

As another example we define `satisfy`. It succeeds when the next input item satisfies a given predicate, and returns the input item if successful.

```
satisfy  ::  (a -> Bool) -> Parser [a] a
satisfy p xs = [ (r,xs') | (r, xs') <- item xs, p r]
```

Now we can define `literal`, which succeeds when the next item has a particular value, and `seq`, which matches two argument parsers and returns their combined result. The definition of `seq` uses the lambda abstraction (`\arg -> expr`) to define a new parser and return the correct result.

```
literal :: Eq i => [i] -> Parser [i] i
literal x = satisfy (==x)

seq :: Parser i a -> Parser i b -> Parser i (a,b)
p1 `seq` p2 = p1 `into` (\a -> p2 `into` \b -> okay (a,b))
```

The fact that parsers are functions means that we can combine them easily and write other higher order functions that operate on them. For example, we can define the function `oneof` which succeeds when one parser of a list of parsers suceeds and which returns the result of that match. It means that we must insert the `orelse` operator between all the parsers. Again, the use of `orelse` means that all possible successes are collected:

```
oneof   :: [Parser a b]  -> Parser a b
oneof l = foldr1 orelse l
```

2.3. Parsers as monads

Given our definitions we can define parsers as monads [Wadler92]. Conceptually, monads allow us to make a distinction between a value and the computation that produces that value. Specifically, a monad `m a` denotes a computation of type `m` that, when evaluated, will produce a value of type a. To define parsers as (various kinds of) monads we have to define a number of functions. Using the monad constructor classes [Jones93] defined in Gofer, this results in:

```
instance Functor (Parser i) where
                 map  f p  = p `into` (okay.f)
instance Monad   (Parser i) where
                 result = okay
                 bind = into
```

The main advantage of treating parsers as monads is that we can glue together parsers using the functions `bind` and `result`. If we change the implementation of parsers, our definitions can remain unchanged, as long as the new implementation can again be mapped to a monad by defining implementations for `bind` and `result`. By adding a few more definitions, for a zero element and for the concatenation:

```
instance Monad0    (Parser i) where
                 zero    = fail
instance MonadPlus (Parser i) where
                 (++)    = orelse
```

we can use monad-comprehensions to define our parsers. Monad comprehensions are similar to list-comprehensions (in fact, the list is a monad). We illustrate the use monad comprehensions by defining two auxiliary combinators. The first is `serie` which matches a list of parsers in sequence and returns the combined results as a list. The second one is `option`, which matches the optional construct. If the argument parser succeeds, its result is returned (in a list), otherwise the empty list is returned. In fact, these functions operate on any monad of the given class.

```
serie :: Monad m => [m a] -> m [a]
serie [] = result []
serie (p:ps)  = [ a:as | a <- p, as <- serie ps ]

option :: MonadPlus m => m a -> m [a]
option p   =  [ [x] | x <- p] ++ result []
```

3. Using parsers to model processes

The theme of this paper is to use parsers to model office processes. This means that the actions performed by office workers are consumed by a parser which checks whether they correspond to a legal sentence as defined by the grammar for the procedure. We start of by describing our representation of actions. To illustrate the general idea we then give a small first example.

3.1. Parsers of actions

We represent an action as three-tuple. The first element is a label, which (uniquely) identifies an action within a procedure. The second element is the value that was produced in the action. The final element is a tuple of properties associated with this action. In this paper we assume that it holds an actor field (identifying the actor that performed it) and a timestamp field, indicating the time the action was performed or completed.

```
type Label      = String
type TimeStamp  = String
type Actor      = String
type Properties = (Actor, TimeStamp)
type Action     = (Label, Value, Properties)
```

The value associated with an action can either be an elementary value (here we assume Text and Numbers) or composite values, which are either sets of values (represented as lists) or a record, which is a collection of name value pairs, in our case represented by actions. Note that in this report we do not check whether a set-value is actually a set.

```
data Value      = Number Int
                | Text String
                | Set [Value]
                | Rec [Action]
```

For simplicity, we define a couple of shorthands that extract (particular) values out of actions:

```
actval :: Action -> Value
actval (l,v,p)   = v

setval (l,(Set v), p) = v
numval (l,(Number v), p) = v
txtval (l,(Text v), p) = v
recval (l,(Rec v), p) = v
```

Each elementary parser in our procedures is assumed to result in an action object that was parsed. To parse input actions, we can use the existing combinators. For example, to define a new parser that matches an action with a given label, we can define the following:

```
actl :: Label -> Parser [Action] Action
actl l = satisfy (\(x,d,(u,t)) -> x == l)
```

Recall that `satisfy` is passed the next input item, which in this case is an action.

We assume henceforth that all of our parsers return action objects, so any parse function has a type definition similar to `actl` above. We glue the resulting action objects together to form the dossier that is produced as a result. Composite parsers must thus also return action objects. To do this, we define a few functions that turn lists of action objects into other action objects or create new ones:

```
tset,trec :: Label -> [Action] -> Action
tset l x = (l, (Set (map actval x)), ("system", "0"))
trec l x = (l, (Rec x), ("system", "0"))

tnum :: Label -> Int -> Action
tnum l i = (l, (Number i), ("system", 0))
```

Note that the resulting action objects are provided with a default actor and a default time-stamp. Of course it would also be possible to associate a set of actors with an action and collect the actors from the constituent actions. The label can be anything, but in general it will correspond to the name of the non-terminal that invokes it. This allows us to get a direct association between the grammar-rules and the data-objects produced. We can turn these two "action constructors" into action parsers as follows:

```
aset, arec :: Label -> [Parser [Action] Action] -> Parser [Action] [Action]
aset l acts = [tset l v | v <- acts]
arec l acts = [trec l v | v <- acts]
```

Now, for example, we can define a parser for some form which is composed of three fields that have to be filled in in sequence:

```
form = arec "form" (serie[field1, field2, field3])
```

It will return an action labelled with `form` and containing a record with the three actions produced by the field parsers.

3.2. A first example: expense reimbursement

Consider a procedure for reimbursement of travel expenses. It is initiated by a user who has to specify the details of the trip, and in particular of the money that was spent. This specification is then sent to the user's manager, who must approve the reimbursement but who may also refuse it. If approval is obtained, the administration will transfer the money to the employee. Simplifying somewhat we can model the top-level of this procedure as follows:

```
expenseclaim  = arec "expenseclaim"
                  (serie [expenseform, inspect, oneof [reimbursed, refused]])
reimbursed    = arec "reimbursed"  (serie [approved, reimbursement])
```

Filling out the expense form means providing several pieces of information, broken down further into personal information and information on the claim itself, such as the project for which the expenses were made, the amount of travel expenses involved, the conference attendance fee and other costs:

```
expenseform = arec "expenseform" (serie [personal, claim])
personal    = arec "personal" (serie [requester,department,bankaccount])
claim       = arec "claim" (serie [project,travel,conference,other])
```

The other parsers in are modelled as elementary action parsers, i.e.

```
inspect     = actl "inspect"        approved    = actl "approved"
refused     = actl "refused"        requester   = actl "requester"
department  = actl "department"     bankaccount = actl "bankaccount"
project     = actl "project"        travel      = actl "travel"
conference  = actl "conference"     other       = actl "other"
reimbursement = actl "reimbursement"
```

Now we have completed the definition of our procedure, and using Gofer we can run this on input lists of actions. If the action sequence is acceptable, the result will be a (composite) action object holding the results.

4. Constraints and context sensitivity

The first example shows that a parser provides an intuitively appealing description of a procedure. The structure of the process (in terms of ordering of actions) is immediately clear through the use of combinators like `serie` and `oneof`. Furthermore, splitting up the parsing in functions provide a straightforward decomposition mechanism to break the steps down into

smaller ones.

Of course, our example is too simplified for any practical use. On the one hand, it is too strict. For example, it does not allow situations where the form was filled in incorrectly, or where the manager requires extra information. In this case, the requester should be asked to provide a new (or adapted) form with the correct or additional information. This can be solved by adding a check step and optional creation of a new form, e.g. as in:

```
checkedexpenseform  = [ f | x <- expenseform, f <- checkedform x]
checkedform  f      = [ n | c <- review f, x <- expenseform, n <- checkedform x]
                     ++
                      [ f | _ <- accept f]
```

To mimic the flow of information – in this case the right version of the form – we use the capability to pass results of parsers as argument to other parsers.

On the other hand, the grammar is too loose. It allows for actions and thus traces that should not be allowed. First, the expenses in the claim should be numbers. This can be repaired easily by adapting the elementary parsers to check for the type e.g. by using a parser like this:

```
numactl n = satisfy (\(l,v,p) -> l == n && nu v  where nu (Number _) = True
                                                        nu _ = False)
```

Another issue is that approval of the claim should be done by the manager of the person who filed the form. To do this, we must first of all pass the information about the requester to the parsers that encode inspection and the approval or refusal. It means that the expenseform produced earlier should be passed as an argument, e.g. like this:

```
expenseclaim   = arec "expenseclaim" [[e,i,h] |
                   e <- expenseform, i <- inspect e,
                   h <- oneof [reimbursed e, refused e]]
```

Now we can adapt as an example the parser for approval to check whether the action was produced by the right person, i.e. the manager of the requester. It receives the expenseform created earlier as an agument. We assume a function `lookup` that returns tan action object with a particular label in another action object that is a record, and the function `managerof` which returns the manager of a person.

```
approved  f   = satisfy \(l,v,(u,t)) -> l == "approve" && u == managerof r
          where  r = txtval (lookup f "requester")
```

Obviously, this is a very trivial modelling of a "role-constraint". But is also clear how more complex constraints can be modelled using this basic idea. We can use some sort of organizational database containing relations like "manager of". This can then be exploited in more elaborate conditions associated with the parser.

In essence, what we do here is make parsers context sensitive; whether or not the "approved" action is accepted or not, not only depends on the order of the input, but also whether the actor part matches a value obtained earlier in the parse process. A similar approach can be used to handle other issues.

Suppose, for example, that we want to ensure that the total amount that is reimbursed matches the total of the costs produced earlier. First, we have to add the different costs to a total and make it part of the form. This can be done by modifying the parser for claim and extending it with a a calculation of the total:

```
claim = arec "claim" [[p,t,c,o,tnum "total" (sum (map (numval [t,c,o])))] |
                   p <- project, t <- travel, c <- conference, o <-other]
```

Now, in the parsers that handle reimbursement, we can check whether the amount reimbursed matches this total:

```
reimburse f = satisfy (\(l,v,p) -> l == "reimburse" && (nu v) == r where
                 nu (Number v) = v
                 r = numval (lookup f "total")
```

In a similar vein, we can add constraints that check whether the whole procedure is handled within a certain time frame, e.g. when a decision should be made within a specified amount of time after the initiation. Tthe timestamp associated with the first field in the form can be used to check whether the timestamp for the inspection is within the given limit.

5. Dynamic instantiation and permutations

In the previous section we have seen how grammars can model a simple reimbursement process, and how using context sensitivity allows us to express (and check) role-constraints and time-constraints. In this section we extend our formalism somewhat to handle more complex processes.

Suppose we want to model a voting process. A user can initiate a vote by defining the topic, the list of participants that should vote and the list of choices from which they can choose. After this has been done, the participants can cast their votes until some deadline is passed. The top-level looks like this:

```
voting  = arec "voting" [[p,v] | p <- prepare, v <- votes p])
prepare = arec "prepare" (serie [topic,choices,parts,deadline])
```

We assume that `topic` and `deadline` are handled by elementary parsers. The other ingredients of `prepare` are modelled as follows:

```
choices = aset "choices"      (many1 (actl "choice")
parts   = aset "participants" (many1 (actl "participant")
```

Here we use the combinator `many1` which matches one or more occurences of a parser and returns all results in a list:

```
many, many1 :: MonadPlus m => m a -> m [a]
many  p =  [ (f:r) | f <- p, r <- many p] ++ result []
many1 p =  [ (f:r) | f <- p, r <- many p]
```

So far, this is similar to our previous example. It becomes different however, when we want to model the casting of votes. The basic idea is that each participant should cast precisely one vote, and that this vote should contain a choice defined in the set of choices defined initially. For the time being we assume that voting is compulsory. We can do this by "generating" combinations of parsers.

The elementary parser is `vote` which succeeds when a given choice is made by a given participant:

```
vote p c = satisfy (\(l,v,(u,t)) -> l == "vote" && p == u && v == c)
```

Obviously, this can be extended with a constraint on `t` if we want to enforce that all votes should be received before the deadline. Assuming that `f` holds the action object produced in preparation, we get:

```
vote' f p c = satisfy (\(l,v,(u,t)) -> l == "vote" && p == u && v == c && t < dl
                 where dl = txtval (lookup f "deadline"))
```

Given a particular participant p and the action object f produced during preparation, we can generate a parser that accepts one of all possible choices as follows:

```
pvote f p = [ v | v <- oneof [ vote' f p c | c <- setval (lookup f "choices")]]
```

Similarly, we can generate a series of parsers for the set of participants and thus define a first version of the parser for votes:

```
votes' f  = aset "votes" [ v |
            v <- serie [ pvote f (txtval p) |p <- setval (lookup f "participants) ]]
```

The problem with this definition is that we use the serie combinator. The votes by the participants now are expected in a particular order. Of course, this is undesired: we want to accept all possible permutations of votes [Cameron94].

We are looking for a new combinator unordered which will match a list of parsers occurring in arbitrary order. The type of this combinator will be something like:

```
unordered     :: [Parser s a] -> Parser s [a]
```

Defining unordered is conceptually simple. We generate all possible permutations of the list of parsers, put each permutation in a serie, and combine the set of permutations as alternatives using orelse. Lazy evaluation will take care of optimizing this for the particular case needed:

```
unordered ps = oneof (map serie (perms ps))
               where gaps [] = []
                     gaps (x:xs) = xs:(map (x:) (gaps xs))
                     perms [] = [[]]
                     perms l  = [ x:p | (x,r) <- zip l (gaps l), p <- perms r]
```

Note that there is one flaw in this solution: the order of the results is not fixed but depends on the alternative actually found. To correct this, we must add functions to re-order each alternative once it has been matched. We will not discuss this further here.

Using unordered in our example completes the definition of the votes procedure:

```
votes f  = aset "votes" [ v |
           v <- unordered [ pvote f (txtval p) |
                            p <- setval (lookup f "participants)]]
```

Removing the restriction on compulsory voting is simple. We make voting optional by using the option construct:

```
votes f  = aset "votes" [ concat v |
           v <- unordered [ option (pvote f (txtval p)) |
                            p <- setval (lookup f "participants)]]
```

6. Parallel parsers

This voting procedure has shown how easily dynamic instantiation of particular parts of the procedure can be encoded. Combined with the use of permutation parsing (unordered) this seems to provide interesting opportunities. For example, it would seem that we can use this to model a semi-structured conversation process where multiple discussion threads can be created in response to a message.

The top-level of such a procedure could be something like this:

```
econv = arec "conversation" [[m,r] | m <- message, r <- reactions m]
```

We assume that messages are represented by at least three actions, giving the subject, a unique message identifier and a body (represented by elementary parsers):

```
message =  arec "message" (serie [subject,id,body])
```

A reaction to a message is a message, which should refer to the message identifier of the original:

```
reply m   =  arec "message"  (serie [subject, replyto m, id, body])
replyto m = satisfy \(l,v,p) -> l == "replyto" && v == actval (lookup m "id"))
```

Generalizing somewhat, we can say that each reaction to a message leads to a thread: an initial reply and reactions to this reply:

```
thread m = arec "thread" [ [x,r] | x <- reply m, r <- reactions x]
```

Now the only thing left is the modelling of `reactions`. It is a parser that collects arbitrary collection of reactions to a message, each of which may form a thread. A first attempt would be:

```
reactions m  = aset "reactions" (many (thread m))
```

Recall that `many` matches zero or more occurrences of its argument parsers. While this is basically correct (any message can lead to zero or more threads), the overall parser now has one basic flaw. The threads have to occur in order. If we think of the resulting conversation structure as a tree, the tree has to be provided in the input in a depth-first way. Of course, this is not the way in which a conversation normally takes place. What we really want is that messages and threads can be added to the conversation in any order. While one message may be part of a nested thread, the following message may be a reaction to the initial message, etc.

The `unordered` combinator introduced does not provide us with a solution. It handles arbitrary interleavings of its argument parsers, but not of their constituent parsers. This is however what is needed here. What we need is real "parallellism", so that threads can be built-up in arbitrary order.

6.1. The parallel combinator

The problem to solve is to construct a combinator that will allow arbitrary interleavings of its constituents, in a recursive fashion. In fact, we want to extend our formalism that has state and non-determinism with parallellism. To do this, we have to change fundamentally the way in which parsers work. Conceptually, we have to represent parsers as "processes", and allow for arbitrary interleavings of their executions.

We use resumptions to model this behaviour. A parser is a function that after performing one step (e.g. match an input item) delivers a final result or a resumption, a process that can be restarted later and then continues parsing. By using the merge operator we can generate all the possible execution traces of these resumptions. The (free) merge in process-algebra is defined as follows (where `x` and `y` represent composite processes and `a` represents a result):

```
x      ||  y  = x ||_ y + y ||_ x
a      ||_ x  = ax
ax     ||_ y  = a (x || y)
(x+y)  ||_ z  = x ||_ z + y ||_ z
```

To represent parsers in a resumption style, we have to adapt their type definition:

```
type Parser s a = s -> [PStat s a]
data PStat  s a = Done a s
                | Pause s (Parser s a)
```

A parser operates on a state denoted by `s`, and produces a value of type `a`. The parser is a function

that produces a set of resumptions of type PStat. Each resumption holds the resulting state. In addition, a resumption indicates either that the processing is complete (Done) or that more work is to be done (Pause). In the latter case, a new parser is delivered that should still be performed.

We now can define the elementary parsers for this new scheme:

```
fail    ::  Parser a b              okay         :: a -> Parser b a
fail xs  =  []                      okay v xs = [Done v xs]

pause :: Parser a b -> Parser a b
pause p r = [Pause r p]
```

Fail and okay are similar to before. The new elementary parser is pause which delivers a resumption that can be continued later on.

Most of our other definitions do not have to be changed; they are expressed purely in terms of fail and okay or are defined using monads. Since parsers with resumptions can also be defined as monads (using the same definitions as given above), these present no problem. However, combinators into and orelse have to be revised:

```
into :: Parser i a -> (a -> Parser i b) -> Parser i b
(p1 `into` p2) xs = [ bi | pi <- p1 xs,
                           bi <- case pi of
                                 Pause i' p' -> pause (p' `into` p2) i'
                                 Done  a  i' -> pause (p2 a) i' ]

orelse   ::  Parser a i -> Parser a i -> Parser a i
(p1 `orelse` p2) xs = (pause p1 xs ++ pause p2 xs)
```

The general idea here is that after one step is performed, the combinator will return a Pause resumption, and when the whole parsing has been completed, it will return a Done resumption with the final result.

We are now ready to define the parallel combinator. We want this combinator to return the results of its constituents in an ordered fashion (in a tuple), so that we can use it as an operator, e.g.:

```
sample = n1 `par` n2 `par` n3
```

This means that we have to alter the definition of the free merge slightly, to take the ordering of results into account. The resulting definition of the par combinator and the auxiliary functions lmerge and rmerge is given below:

```
par,lmerge,rmerge :: Parser s a -> Parser s b -> Parser s (a,b)
(p1 `par` p2) i = (p1 `lmerge` p2) i ++ (p1 `rmerge` p2) i

(p1 `lmerge` p2) xs = [ bi | pi <- p1 xs,
                             bi <- case pi of
                               Pause i' p' -> pause (p' `par` p2) i'
                               Done  a  i' -> pause (p2 `into`
                                     (\x s -> okay (a,x) s)) i']

(p1 `rmerge` p2) xs = [ bi | pi <- p2 xs,
                             bi <- case pi of
                               Pause i' p' -> pause (p1 `par` p') i'
                               Done  a  i' -> pause (p1 `into`
                                     (\x s -> okay (x,a) s)) i']
```

We also introduce a combinator parl which provides parallellism for a list of parsers:

```
parl ::  [Parser i a] -> Parser i [a]
parl [] = result []
parl (p:ps) = map trf (p `par` parl ps) where trf (a,as) = a:as
```

Note that a parser is no longer a function that directly delivers the parse results. It returns a resumption. This means that we need an auxiliary function to actually do the parsing and transform the resulting resumptions into the real results:

```
flattend p xs = fflat (p xs)
                where   fflat   [] = []
                        fflat ((Done r s):ps)  = [Done r s] ++ fflat ps
                        fflat ((Pause s p):ps) = (fflat (p s)) ++ fflat ps

parse :: Parser i a -> i -> [(a,i)]
parse p xs = map fres (flattend p xs)
                where fres (Done r s) = (r,s)
```

Finally, we can use the definition of `flattened` in the definition of atomic, which allows us to express that a parser cannot be interleaved with others:

```
atomic :: Parser i a -> Parser i a
atomic p = \xs -> flattend p xs
```

We can use `atomic` in our basic parsers. For example, it would be reasonable to claim that the sub-steps of `satisfy` should not be interleaved with others, so that we can express satisfy (using monads) as follows:

```
satisfy p = atomic ([s | s <- item, p s])
```

Also note that we can now create an operator variant of `unordered` as follows:

```
p `unord` q = (atomic p) `parl` (atomic q)
```

Obviously, `unordered` itself could now be expressed in terms of `unord`. However the original definition has the advantage that it does not require the use of resumptions.

6.2. E-mail revisited

Returning to the example of electronic conversations, we can put the parallel combinator to use. Recall that the main problem was to allow for arbitrary interleaving of the building up of threads in reaction to a message. This can now be expressed as follows:

```
econv = arec "conversation" [[m,r] | m <- message, r <- reactions m]

reactions m  = aset "reactions" (threads m)

threads m =     [ ((torec "reaction" [x,r]):t) |  x <- reply m,
                            (r,t) <- reactions x `par` threads m]
          ++  [[]]
```

The use of `par` forks the parallel parsers for arbitrary many threads. Finally, to ensure that messages are still handled as units, we have to use `atomic`.

```
message =  arec "message" (atomic (serie [subject,id,body]))

reply m   =  arec "message" (atomic (serie [subject, replyto m, id, body]))
```

Again, we note that the example can be extended easily in many ways. For example, we could identify a group of people involved in the conversation, and enforce that only they can add messages. Assuming that each of them can react at most once to each message, we could then generate "reaction slots", similarly to the voting process defined earlier.

7. Shared state

The dynamic instantiation properties combined with the parallellism introduced above provide new dimensions. For example, if we think of a software process model describing the organized modification of a software system, we can imagine instantiating multiple parallel sub-tracks that modifiy different modules. Each of these can be instantiated with a particular engineer and a particular module.

In fact, such a process model has been defined as a test-case for software process modelling and process programming environments (the ISPW-6 and ISPW-7 examples [Heimbigner91]). In this section we will explore this example further, focusing on issues not addressed in previous examples. Again our emphasis is on process modelling. The definition of the ISPW cases contains many operational issues, such as the medium on which documents are represented or notifying people of completion of certain steps. We will not consider these issues here.

The basic idea of the ISPW examples is to model the handling of a change request on a software system. It begins with the assignment of tasks to engineers and the scheduling of activities in time. The real work is done in steps that adapt and review the design, change and compile the code of a module, adapt the test-plans and test package and that test the changed unit until the test results are satisfactory. The definition of the case contains a lot of parallel activities. The adaptation of the design may be performed in parallel with the adaptation of the code and the adaptation of the test information. In parallel with the overall work-process there is an activity called "monitoring" which can change the assignment of tasks to engineers and reschedule deadlines.

Modelling of the ISPW example requires one extension of our formalism. It involves the sharing of information between parsers. If we have true parallellism, our existing approach of passing results of one parser as arguments to others does not suffice to share information. For example, the task assignment produced as part of the preparation phase may be changed on the fly in the monitoring activity. So, we need to introduce a concept of "state" which can be shared transparantly among all parsers.

7.1. Modelling a shared environment

Recall the type definition of a parser:

```
type Parser i a = i -> [(a,i)]
```

A parser is a function that transforms a state into a list of value, state pairs. Until now we have implicitly assumed that the state is a list of items, but there is no reason why the state cannot be a more complex entity, as long as it provides the means to get the next "item". We thus abstract the characteristics of the parsing state into a constructor class. Any type for which we can define the two functions `next` and `isempty` can act as a parsing state:

```
class ParsingState s where
        next ::  s a -> (a,s a)
        isempty :: s a -> Bool
```

Obviously, the list type can satisfy this requirement easily:

```
instance ParsingState [] where
        isempty [] = True
        isempty (i:is) = False
        next (i:is) = (i,is)
```

The only parser that really must change is `item` since it assumed a list to represent the state. Its new definition is:

```
item ::  ParsingState t => Parser (t a) a
item s =  if (isempty s) then fail s else v (next s)
                where v (x,ys) = okay x ys
```

Note that the type-definition of our other parsers will also change, to reflect that we assume a type that satisfies `ParsingState`.

Now we introduce another class, which is a `ParsingState` but also holds an environment, a list of associations between labels and values. For simplicity's sake we use an `Action` object (with a Record value) to represent such an association:

```
class ParsingState s => EnvParsingState s where
        addenv ::  s a -> Action -> s a
        fndenv ::  s a -> Label -> Value
```

A possible implementation of this class is:

```
data    PEStat a = PES [a] Assoc

instance ParsingState PEStat where
        next (PES (i:is) x) = (i, PES is x)
        isempty (PES [] _) = True
        isempty (PES x _) = False

instance EnvParsingState PEStat where
        addenv  (PES a (l, Rec v, props)) as = (PES a (l, Rec (as:v), props))
        fndenv  (PES _ x) l = lookup x l
```

Defining parsers that operate on this type of parsing state provides us with shared state. But to obtain access to the state, we have to add a few extra definitions. First, we introduce an elementary parser called `statep` which takes a function that transforms the state and returns it, and a parser `stateq` which succeeds when the state matches a certain predicate:

```
statep :: (t -> t) -> Parser t t
statep p s = let r = (p s) in okay r r

stateq :: (t -> Bool) -> Parser t t
stateq p s = if (p s) then okay s s else fail s
```

Now we can define functions to update the environment (eadd) and lookup values (`elup`). Also we define a parser combinator aenv which applies a parser and stores the resulting action in the environment.

```
elup :: EnvParsingState t => Label -> Parser (t a) Value
elup l = [fndenv x l | x <- statq (\_ -> True) ]

eadd :: EnvParsingState t => Action ->  Parser (t a) Value
eadd a = [ a | _ <- statep (\s -> addenv s a) ]

aenv :: EnvParsingState t =>  Parser (t a) Assoc -> Parser (t a) Assoc
aenv p = [ a | a <- p, _ <- eadd a]
```

With these definitions, we can write parsers that add an association to the environment which other parsers can retrieve later on.

7.2. The ISPW example

Now we return to the ISPW example and illustrate how the mechanisms introduced above can be put to use. The parsers that we see now have basically the following type:

```
type EnvActParser = Parser (PEStat Action) Action
```

The toplevel parser for the ISPW model can be defined as follows:

```
changeprocess = arec "changeprocess" [ [i,w,r,f] |
                i <- prepare, [w,r] <- parl [work, monitor], f <- testedchange ]
```

`Prepare` deals with the initialization of the process. Conforming the example, it involves the receipt of the arguments to the process (description of the change-request, authorization of the configuration control board), the definition of the material involved (design document, source module, test plan and package), the identification of the team members (manager, engineers), the allocation of people to roles (design engineer responsible for the design and the coding, quality engineer responsible for the test specifications, review team that will perform the design review) and the scheduling of activities (date for review, date for termination).

The general outline of the definitions is as follows. For brevity we only list the definition of non-terminal action parsers. We assume that the relevant "terminal" action parsers (e.g. `code`, `design`, `reviewdate`, `completiondate`, etc.) store their results in the environment. Also we have not included "role-checks" for these actions. Typically, the assignment and scheduling will be done by the project manager and this should be checked.

```
prepare   = arec "prepare"   (serie [arguments,material, team, assign, schedule])
arguments = arec "arguments" (parl [changerequest, authorization ])
material  = arec "material"  (parl [design, code, testsplan, unittest])
team      = arec "team"      (parl [manager, engineers])
schedule  = arec "schedule"  (parl [reviewdate, completiondate])
assign    = arec "assign" [d | d <- parl [designengineer,qaengineer,reviewteam],
                               _ <- checkroles]

engineers = aenv (aset "engineers" (many1 engineer))

designengineer = bindengineer "designengineer"
qaengineer = bindengineer "quaengineer"

bindengineer role = atomic (aenv [ x |
                      ps <- (elup "engineers"),
                      x <- oneof [ satisfy  (\(l,v,_) -> l == role && v == p)
                                 | p <- setval ps ]])
```

Note that we can easily express the fact that the two engineer roles should be assigned to members of the team by generating alternative parsers. The parser `checkroles` has been added to the definition of `assign` to model additional checks for role-bindings. Typically, we might want to enforce that the quality engineer and the design engineer are different people, for example:

```
checkroles = [d | q <- elup "quaengineer", d <- elup "designengineer", d /= q ]
```

The monitor activity runs in parallel with the actual work. Its main purpose is to facilitate rescheduling of tasks and reassignment of roles. This can now be modelled very easily; it allows for reoccurrence of some of the actions defined above at a later stage. Again, for simplicity's sake checks for the fact that the manager can only perform these actions have been excluded:

```
monitor = arec "monitor" (many (oneof [reassign,reschedule]))

reassign = [r | r <- oneof [designengineer, qaengineer, reviewteam],
                _ <- checkroles]
reschedule = oneof [reviewdate, completiondate]
```

Since existing parsers add the actions to the environment, the results are available for other parsers that run in parallel. Note that the binding of a role is an atomic action to prevent interleaving between consumption of the action and the update of the environment.

Now we turn to the real work on the software system. It consists of three parallel activities:

```
work =  arec "work" (parl [newdesign, newcode, newtest])
```

Production of the new design implies modification of the design document and a review. Once the review has been completed sucessfully, the new design is accepted. Otherwise, modifications are in order and a new review will occur. Again, for the sake of brevity, we have excluded issues like time-checks (e.g. the fact that the review should occur on a given date). Note that the definitions include the collection of statistics, e.g. the effort that was involved in the review.

```
newdesign  = aenv [ f | x <- editdesign, r <- review x, f <- revwddesign x]

revwddesign f = [ n | _ <- revise, n <- newdesign]
                ++
                [ f | _ <- accept]

editdesign = atomic [x| p <- elup "designengineer",
                        x <- satisfy (\(l,v,(u,t)) -> l == "design" && u = txtval p)]

accept  = arec "accept" [ [e]   | a <- actl "accept", e <- actl "effort"]
revise  = arec "revise" [ [c,d] | c <- actl "comments", d <- actl "nrofdefects"]
```

Note that we use `atomic` in the definition of `editdesign`. The reason for this is somewhat intricate. Since looking up a value in the environment (`elup`) is modelled as a separate parser, the two parsers that make up `editdesign` may occur interleaved with the `monitor` parser that captures reassignment. If `atomic` were not used, a reassignment that happens just before the adaptation of the design might be missed. In particular, the `elup` parser would have succeeded, returning the old binding of the role, the `monitor` parser would succeed, giving a new role binding in the environment, and the `design` action (performed by the person just bound to the role) would be refused, since `satisfy` expects an old value. The use of `atomic` prevents this, by combining the environment lookup and the subsequent satisfy into one step. Because of a similar argument, the binding of a person to a role is also performed atomically (see above).

The development of the source code is modelled similarly. It includes a compilation step, and only if no errors during compilation occur will the new code be accepted. The results produced by the compilation step can be quite elaborate (options, multiple object code modules for different targets, etc).

```
newcode = aenv [ c | x <- editcode, r <- compile, c <- compcode x]

editcode = [ x | p <- elup "designengineer",
                 x <- satisfy (\(l,v,(u,t)) -> l == "code" && u = textval p)]

compcode c = [ n | x <- compileerrors, n <- newcode]
             ++
             [ trec "compiledcode" [c, r] | r <- compiledprogram]

compiledprogram = arec "compiled" (atomic (serie [options, objcode, compinfo]))
```

The ISPW specification states that the development of new code cannot be completed until the new design has been completed. This involves synchronization between parallel processes. While this can be solved using the shared environment, we have excluded the definition here. The same holds for the updating of the test package and the test plan, because they are similar to the definitions above (but performed by the qaengineer).

We finish the example with the final step of the process, applying the new test to the new code. The idea is to have a test-run and to check the results. These can indicate that the test set should be

adapted (if coverage analysis shows that the coverage is less than 90 percent) and/or that the program is not correct (test errors). If neither of these problems occur, the change has been completed.

```
testedchange = arec "testedchange"  serie [runtest, handletest]
handletest = arec "revise" [ tc |
                _ <- oneof [ serie [testfail, codefail, reviseboth ],
                             serie [testok, codefail, newcode],
                             serie [testfail, codeok, newtest]]
                tc <- testedchange]
            ++ arec "success" (serie [testok, codeok])

reviseboth = arec "adapt" (parl [newtest, newcode])
testfail = satisfy (\(l,v,p) -> l == "coverage" && (nv v) < 90)
testok = satisfy (\(l,v,p) -> l == "coverage" && (nv v) >= 90)
codeok = satisfy (\(l,v,p) -> l == "nrerrors" && (nv v) == 0)
codefail = satisfy (\(l,v,p) -> l == "nrerrors" && (nv v) > 0)
```

8. Conclusions and evaluation

This paper has presented some of the research into office process support in the Ariadne project at Utrecht University. An important aspect of our appoach is to separate the operational issues surrounding process enactment from the conceptual description of a process as a space of legal actions or events. In our view, a process model should provide enough information to decide whether a particular sequence of events that takes place is legal. It should not be "muddled" with issues that describe how the actual performance of these activities is generated. We believe that making this distinction is crucial to allow for different enactment styles that use the same process description.

In the previous sections we have illustrated how functional parsers can be used to encode this conceptual view of processes. We have applied parser combinators operating on strings of actions to model several non-trivial processes. Starting with a simple model of parsers as non-determinstic state transformers, we have increased the expressive power, partly by changing the underlying semantics of parsers. The final version includes state, non-determinism and parallellism. The examples show how these provisions, together with context-sensitivity, allow us to describe role- and binding constraints, time- and value constraints, dynamic instantiation of parsers, and permutations and parallel activities. A nice side effect of encoding these parsers in a functional language is that the models are executable. All the examples are stylized and abbreviated versions of Gofer programs that can be run on collections of actions. Also, it is easy to extend the set of available constructs.

The resulting process models are reasonably simple and elegant despite the underlying complexity (though it is arguable whether "naive end-users" will be able to use the model in its current form). This is mainly due to the use of monads, which separate the value and the computation that produces the value. The availability of function abstraction makes it possible to collect complex sub-models and to reuse these in different places. Thereby we can structure the process model and maintain an overview.

Due to the simplicity of the models, they are also relatively simple to adapt. For example, by replacing a "terminal action" by a "non-terminal" we can refine the action decomposition. Similarly we can add more constraints on the structure of the values that are produced. Note however, that even if such structure is not imposed, the actual data itself can still be structured in using the tree-like record structures that also encode actions.

On the other hand, we can generalize a process model by removing constraints or discarding detailed action descriptions. This may prove useful for exception handling. For example, the simplest process model is something like this:

```
any = arec "any" many1 (satisfy (\x -> True))
```

It matches any action by any user at any time with any associated data. Note however, that this model is not useless. It imposes an order on the actions that occur and collects the results in the process' data structure.

Obviously, we can add structure to such a model in several ways. For example, we can enforce that all actions are performed by members of a group, or that they should produce values of a certain type. In this way we can encode a turn taking protocol for a group working on a shared object. In a similar vein, other semi-structured processes can be created incrementally.

8.1. Future work

Although the approach presented here is promising, there are some problems and open issues that remain to be adressed. Here we indicate some of the points that we hope to investigate in the near future.

- The fact that the the process modelling language is free from enactment issues means that other facilities should encode these. What should these look like? Can we use processes as defined here to encode these policies? For example can the binding of an action to an actor that should perform it be modelled as a process? There may be a similarity here with the notions of abstract syntax and concrete syntax used in certain generic language environments such as the Synthesizer Generator.

- In principle, the parsers defined above could be used directly as "conformance checkers". By connecting a Gofer implementation to the outside world, we could translate events into actions and let the parser handle them. However, currently the parsers expect the complete set of actions to be available. In a real-life setting we would like to give feedback immediately, e.g. to tell the user that a particular action is not acceptable. To do so, the parsers have to be incremental, indicating whether there are any possible continuations after parsing one action, and perhaps even indicating which actions should occur next.

- The parallel combinator introduces some new problems. One particular issue is synchronization. In the ISPW example for instance, the fact that one parallel activity has reached a certain state (the design is reviewed) is of relevance for another activity (code development). One way in dealing with this is by using updates to the shared environment. A slightly more elegant way may be to allow one activity (the design review in this case) to create actions which are merged with the sequence of incoming events by users and parsed by the other activities. Another alternative would be to create a parser which is part of both parsing threads and which succeeds only when it finds the right input at the right time within both threads. Clearly more research is needed to find a natural modelling of synchronization.

- While parse-results are handled nicely, the handling of arguments and state is not satisfactory. The environment concept introduced in the previous section hides potential dependencies among (parallel) parsers that actually should be visible e.g. in their interface definition. Also, it is not possible yet to elegantly express constraints on parsers that propagate downwards, e.g. when we want to state that all the actions involved in the modification of a source module should be done by a specific actor. Clearly, a better notation that separates the viewpoints on a process (e.g. data vs. actions) and that allows better expression of dependencies is needed. Of course, this notation could be mapped to the parsers described above. Inspiration could be taken from the inherited and synthesized attributes and the semantic functions in attribute

grammars or the description of events in LOTOS [Brinksma86]. The hierarchical and functional process modelling language described by Katayama [Katayama89] provides a possible approach to integrate attributes with functional processes.

- While the parsers provide an elegant framework for describing processes, we also want to be able to analyse processes and reason about them. For example, we want to see whether a process can terminate in time, or whether consistent role-bindings are possible. Some initial work in this area has been done [Boerboom94, Florijn94c], but further study of the available work on language and parsing theory and semantics is needed to provide definite clues as to how and to what extent this can be done. For example, the relationship with two-level grammars and context-sensitive languages remains to be explored.

- The fact that users can change procedures on the fly to handle exceptions may interfere with our assumption that the actions that occurred in a process should conform to the procedure. While a "re-parse" of all actions is clearly a trivial way to see whether old actions conform to a new procedure, some more advanced support (mapping or collecting actions into other actions, for example) will be needed. A formal notion of equivalence or bisimulation [Baeten90] is similarly desired.

- The fact that we can add and remove structure along several dimension suggest a mechanism for inheritance or specialization of process models. In a sub-model we can then add additional constraints, while benefiting from the design already encoded in the "super-model". In such a case, exception handling may even be a matter of considering a process on a higher level of abstraction, or in other words, as an instance of a super-class. A simple process like "any" defined above could be the root of the hierarchy. While this idea seems viable, the technical and conceptual effects remain to be explored.

- The use of monads in the implementation of parsers could be further extended. For example, the fact that a parser produces a list of successes is but one instance of the more general case where a parser can be parametrized with a monad:

  ```
  type Parser s m a = s -> m (a, s)
  ```

  In fact, a parser and a parsing state can be seen as specific instances of a more general "state-transformer monad". Modelling it as such may lead to further simplification.

Acknowledgements

Doaitse Swierstra is the initiator of the Ariadne project. He suggested the use of parser combinators to encode the grammars and outlined the unordered combinator. Jeroen Fokker provided a helpful introduction to parser combinators. Erik Meijer provided crucial advice on monads, the parallel combinator and the introduction of state.

References

[Ait-Kaci86] Hassan Ait-Kaci, "Type Subsumption as a Model of Computation," in *Proceedings 1st International Workshop on Expert Database Systems*, Larry Kerschberg (Ed.), The Benjamin/Cummings Publishing Company, Inc., 1986.

[Asch92] Wim van Asch, "JAMES – a Journalizing Administrative Message Exchange System," Master's thesis, Utrecht University; Department of Computer Science, August 1992.

[Baeten90] J. Baeten (Ed.), *Applications of Process Algebra*, vol. 17, Cambridge Tracts in Theorethical Computer Science. Cambridge University Press, 1990.

[Bandinelli93] Sergio Bandinelli and Alfonso Fuggetta, "Computational Reflection in Software Process Modelling: the SLANG approach," in *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, 1993.

[Boerboom94] Benno Boerboom, "Primitiva voor het modelleren van samenwerkingsprocessen," Master's thesis, University of Utrecht, Department of Computer Science, Utrecht, the Netherlands, February 1994.

[Borenstein92] Nathaniel S. Borenstein, "Computational Mail as Network Infrastructure for Computer-Supported Cooperative Work," in *CSCW'92 - Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work*, Toronto, Canada, 1992, pp. 67–74.

[Brinksma86] Ed Brinksma, "A tutorial on LOTOS," in *Protocol Specification, Testing and Verification V*, M. Diaz, Ed. Elsevier Science Publishers (North-Holland), 1986.

[Cameron94] Robert D. Cameron, "Extending Context Free Grammars with Permutation Phrases," *ACM Letters on Programming Languages and Systems*, vol. 2, no. 1-4, March-December 1994.

[Clemm89] Geoffrey M. Clemm, "Replacing Version-Control with Job-Control," in *Proceedings 2nd International Workshop on Software Configuration Management*, ACM SIGSOFT Software Engineering Notes, vol. 17, Princeton, New Jersey, November 1989, pp. 162–169.

[Conklin88] Jeff Conklin and Michael L. Begeman, "gIBIS: A Hypertext Tool for Exploratory Policy Discussion," Technical Report STP-082-88, MCC, Software Technology Program, March 1988.

[Croft88] W.B. Croft and L.S. Lefkowitz, "Using a Planner to Support Office Work," in *Proceedings of the 1988 Conference on Office Information Systems (ACM SIGOIS Bulletin, Vol. 9, Nr. 2, 3)*, Robert B. Allen (Ed.), Palo Alto, California, March 1988, pp. 55–62.

[Curtis87] Bill Curtis, Herb Krasner, Vincent Shen, and Neil Iscoe, "On Building Software Process Models Under the Lamppost," in *Proceedings 9th International Conference on Software Engineering*, Monterey, California, March 1987, pp. 96–103.

[Curtis88] Bill Curtis, Herb Krasner, and Neil Iscoe, "A Field Study of the Software Design Process for Large Systems," *Communications of the ACM*, vol. 31, no. 11, November 1988.

[Florijn94a] Gert Florijn, *Workflow management - a Limited View on Office Processes*, Position paper for the CSCW'94 workshop on workflow and office information systems, August 1994 .

[Florijn94b] Gert Florijn, *Ariadne: Computer Support for Hybrid Office Processes*, Working Paper, University of Utrecht, March 1994.

[Florijn94c] Gert Florijn and Benno Boerboom, *Modelling Constructs for Hybrid Office Processes*, Working Paper, University of Utrecht, June 1994.

[Gruhn91] Volker Gruhn, "Analysis of Software Process Models in the Software Process Management Environment MELMAC," in *Software Engineering Environments*, Aberystwyth, Wales, March 1991.

[Gulla94] Jon Atle Gulla and Odd Ivar Lindland, "Modelling Cooperative Work for Workflow Management," in *Proceedings of the 6th International Conference on Advanced Information Systems Engineering*, Lecture Notes in Computer Science, Springer Verlag, Utrecht, the Netherlands, June 1994.

[Hammainen90] Heikki Hämmäinen, Eero Eloranta, and Jari Alasuvanto, "Distributed Form Management," *ACM Transactions on Information Systems*, vol. 8, no. 1, pp. 50–76, January 1990.

[Harrison90] William H. Harrison, Harold Ossher, and Peter F. Sweeney, "Coordinating Concurrent Development," in *CSCW'90 - Proceedings of the 1990 Conference on Computer-Supported Cooperative Work*, Los Angeles, California, October 1990, pp. 157–168.

[Heimbigner91] Dennis Heimbigner and Marc Kellner, *Software Process Example for ISPW-7*, Anon. FTP from ftp.cs.colorado.edu, /pub/cs/techreports/ISPW, August 1991, .

[Heimbigner92] Dennis Heimbigner, "The ProcessWall: A Process State Server Approach to Process Programming," in *Proceedings of the 5th ACM SIGSOFT Symposium on Software Development Environments*, Herbert Weber (Ed.), Tyson's Corner, VA, December 1992.

[Hirschheim85] R.A. Hirschheim, *Office Automation – Concepts, Technologies and Issues.* Addison Wesley, 1985.

[Hofstede89] A.H.M. ter Hofstede, T.H. Verhoef, S. Brinkkemper, and G.M. Wijers, "Expert-based support of Information Modelling," Technical Report RP/im-89/7, Software Engineering Research Centre, October 1989.

[Hutton92] Graham Hutton, "Higher-Order Functions for Parsing," *Journal of Functional Programming*, vol. 2, no. 3, July 1992.

[Jones93] Mark Jones, "A System of Constructor Classes: Overloading and Implicit Higher-Order Polymorfism," in *Proceedings FPCA'93*, 1993.

[Kaiser93] Gail E. Kaiser, Steven S. Popovich, and Israel Z. Ben-Shaul, "A Bi-Level Language for Software Process Modelling," in *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, Maryland, 1993.

[Kaplan91] Simon M. Kaplan, Alan M. Carroll, and Kenneth J. MacGregor, "Supporting Collaborative Processes with ConversationBuilder," in *Proceedings of the 1991 Conference on Organizational Computer Systems (ACM SIGOIS Bulletin Vol. 12, Nr. 2,3)*, Peter de Jong (Ed.), Atlanta, Georgia, November 1991.

[Kaplan92] Simon M. Kaplan, William J. Tolone, Douglas P. Bogia, and Celsina Bignoli, "Flexible, Active Support for Collaborative Work with ConversationBuilder," in *CSCW'92 - Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work*, Toronto, Canada, November 1992, pp. 378–385.

[Karbe90] B. Karbe and N. Ramsperger, "Influence of Exception Handling on the Support of Cooperative Office Work," in *Proceedings of the IFIP WG 8.4 Conference on Multi-User Interfaces and Applications*, Simon Gibbs and Alex A. Verrijn-Stuart (Eds.), Heraklion, Crete, Greece, 1990.

[Katayama89] Takuya Katayama, "A Hierarchical and Functional Software Process Description an its Enaction", in Proceedings of the 11'th International Conference on Software Engineering, Pittsburgh, May 1989.

[Kreifelts91] Thomas Kreifelts, Elke Hinrichs, Karl-Heinz Klein, Peter Seuffert, and Gerd Woetzel, "Experiences with the DOMINO Office Procedure System," in *Proceedings of the Second European Conference on Computer-Supported Cooperative Work*, L. Bannon, M. Robinson, and K. Schmidt (Eds.), Amsterdam, the Netherlands, September 1991, pp. 117–130.

[Lai88] Kum-Yew Lai, Thomas W. Malone, and Keh-Chiang Yu, "Object Lens: a Spreadsheet for Cooperative Work," *ACM Transactions on Office Information Systems*, vol. 6, no. 4, pp. 332–353, October 1988.

[Lehman87] M.M. Lehman, "Process Models, Process Programs, Programming Support," in *Proceedings 9th International Conference on Software Engineering*, Monterey, California, March 1987, pp. 14–16.

[Leymann94] F. Leymann and W. Altenhuber, "Managing Business Processes as an Information Resource," *IBM Systems Journal*, vol. 33, no. 2, 1994.

[Medina-Mora92] Raúl Medina-Mora, Terry Winograd, Rodrigo Flores, and Fernando Flores, "The Action Workflow Approach to Workflow Management Technology," in *CSCW'92 - Proceedings of the ACM 1992 Conference on Computer-Supported Cooperative Work*, Toronto, Canada, November 1992, pp. 281–288.

[Nierstrasz93] Oscar Nierstrasz, "Regular Types for Active Objects," in *Proceedings of the 1993 OOPSLA Conference*, Andreas Paepcke (Ed.), Washington D.C., USA, 1993.

[Osterweil87] Leon Osterweil, "Software Processes are Software Too," in *Proceedings 9th International Conference on Software Engineering*, Monterey, California, March 1987, pp. 2–13.

[Pemberton91] Steven Pemberton and Lon Barfield, "The MUSA Design Methodology," Technical Report 91/12, Software Engineering Research Centre, December 1991.

[Pierce91] R.H. Pierce and J. Smith, "The ARISE Process Modelling System," in *Software Engineering Environments*, Aberystwyth, Wales, March 1991.

[Rose92] Thomas Rose, Carlos Maltzahn, and Matthias Jarke, "Integrating Object and Agent Worlds," in *Proceedings of the 4th International Conference on Advanced Information Systems Engineering*, Lecture Notes in Computer Science, vol. 593, P. Loucopoulos (Ed.), Springer Verlag, Manchester, UK, May 1992, pp. 16–32.

[Shepherd90] Allan Shepherd, Niels Mayer, and Allan Kuchinsky, "Strudel - An Extensible Electronic Conversation Toolkit," in *CSCW'90 - Proceedings of the 1990 Conference on Computer-Supported Cooperative Work*, Los Angeles, California, October 1990, pp. 93–104.

[Suchman87] Lucy Suchman, *Plans and Situated Actions*. Cambridge University Press, 1987.

[Verhoef93] T.F. Verhoef, "Effective Information Modelling Support," Ph.D. thesis, Technical University of Delft, Delft, the Netherlands, 1993.

[Wadler90] P. Wadler, "Comprehending Monads," in *ACM Conference on LISP and Functional Programming*, Nice, France, June 1990.

[Wadler92] P. Wadler, "The Essence of Functional Programming," in *19th Annual Symposium on Principles of Programming Languages*, Santa Fe, New Mexico, January 1992.

[Yasumoto94] Keiichi Yasumoto, Teruo Hiashino, and Kenichi Taniguchi, "Software Process Description using LOTOS and Its Enaction," in *Proceedings of the 16th International Conference on Software Engineering*, Sorrento, Italy, May 1994.

[Zucker91] Jean-Daniel Zucker, "ALF: An Advanced Software Process Oriented Engineering Framework," in *Software Engineering Environments*, Aberystwyth, Wales, March 1991.