On Specifying and Visualising Long Running Empirical Studies

Peter Y. H. Wong   Jeremy Gibbons

Abstract

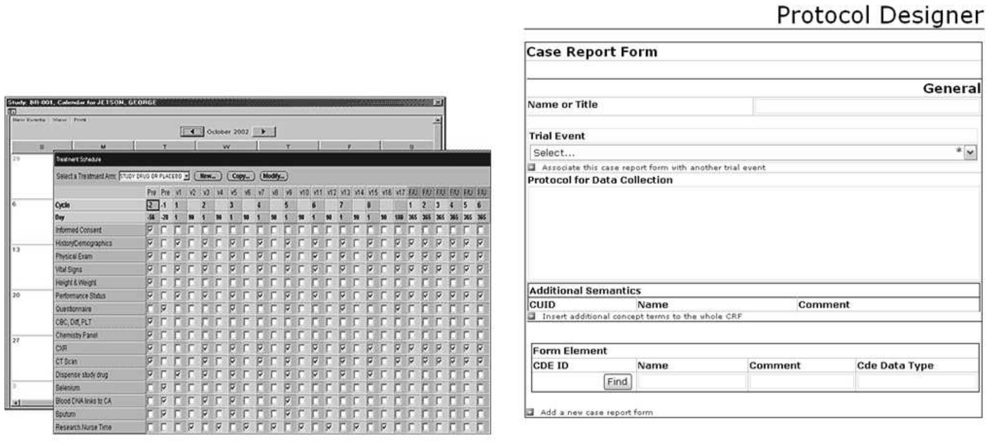
In this paper we describe a graphical approach to formally specifying temporally-ordered activity routines designed for calendar scheduling. We introduce a workflow model *OWorkflow*, for constructing specifications of long running empirical studies such as clinical trials in which observations for gathering data are performed at strict specific times. These observations, either manually performed or automated, are often interleaved with scientific procedures, and their descriptions are recorded in a calendar for scheduling and monitoring to ensure each observation is carried out correctly at a specific time. We also describe a bidirectional transformation between *OWorkflow* and a subset of Business Process Modelling Notation (BPMN), by which graphical specification, simulation, automation and formalisation are made possible.

1 Introduction

A typical long-running empirical study consists of a series of scientific procedures interleaved with a set of observations performed over a period of time; these observations may be manually performed or automated, and are usually recorded in a calendar schedule.

An example of a long-running empirical study is a clinical trial, where observations, specifically case report form submissions, are performed at specific points in the trial. In such examples, observations are interleaved with clinical interventions on patients; precise descriptions of these observations are then recorded in a *patient study calendar* similar to the one shown in Figure 1(a). Currently study planners such as trial designers supply information about observations either textually or by inputting textual information and selecting options on XML-based data entry forms [2], similar to the one shown in Figure 1(b). However, the ordering constraints on observations and scientific procedures are complex, and a precise specification of this information is time consuming and prone to error. We believe the method of specification may be simplified and improved by allowing specifications to be *built formally and graphically, and visualised* as workflow instances.

Workflow instances are descriptions of a composition of activities, each of which describes either a manual task or an application of a program. One of the prominent applications of workflow technology is business processes modelling, for which the Business Process Modelling Notation (BPMN) [11], adopted by Object Management Group [10], has been used as a modelling language. Recent research [12, 15, 13] has also allowed business processes modelled as BPMN



Study: BR-001, Calendar for JETSON, GEORGE

New Events | View | Print

October 2002

S M T W T F S

Treatment Schedule

Select a Treatment Arm: STUDY DRUG OR PLACEBO New... Copy... Modify...

	Pre	Pre	v1	v2	v3	v4	v5	v6	v7	v8	v9	v10	v11	v12	v13	v14	v15	v16	v17	F/U	F/U	F/U	F/U	F/U	F/U
Cycle	-2	-1	1		2		3		4		5		6		7		8			1	2	3	4	5	6
Day	-56	-28	1	90	1	90	1	90	1	90	1	90	1	90	1	90	1	90	180	365	365	365	365	365	365
Informed Consent	✓																								
History/Demographics	✓		✓		✓		✓		✓		✓		✓		✓		✓		✓	✓	✓	✓	✓	✓	✓
Physical Exam	✓		✓		✓		✓		✓		✓		✓		✓		✓		✓	✓	✓	✓	✓	✓	✓
Vital Signs	✓		✓		✓		✓		✓		✓		✓		✓		✓		✓	✓	✓	✓	✓	✓	✓
Height & Weight	✓																								
Performance Status	✓		✓		✓		✓		✓		✓		✓		✓		✓		✓	✓	✓	✓	✓	✓	✓
Questionnaire		✓					✓				✓				✓				✓						
CBC, Diff, PLT	✓																								
Chemistry Panel	✓																								
CXR	✓				✓		✓		✓		✓		✓		✓		✓		✓	✓	✓	✓	✓	✓	✓
CT Scan	✓				✓		✓		✓		✓		✓		✓		✓		✓	✓	✓	✓	✓	✓	✓
Dispense study drug	✓		✓		✓		✓		✓		✓		✓		✓		✓								
Selenium		✓					✓				✓														
Blood DNA links to CA		✓			✓		✓				✓														
Sputum		✓			✓		✓				✓														
Research Nurse Time				✓		✓		✓		✓		✓		✓		✓		✓							

Protocol Designer

Case Report Form

General

Name or Title

Trial Event

Select...

Associate this case report form with another trial event

Protocol for Data Collection

Additional Semantics

CUID	Name	Comment

Insert additional concept terms to the whole CRF

Form Element

CDE ID	Name	Comment	Cde Data Type
Find			

Add a new case report form

(a) (b)

Figure 1: (a) A screen shot of the patient study calendar [3], (b) XML-based data entry forms [2]

diagrams to be translated into executable processes in the Business Process Execution Language (WS-BPEL) [1], the "de facto" standard for web service compositions. Furthermore, a relative timed semantics have been defined for BPMN, in the languages of Communicating Sequential Processes (CSP) [16]; these allow BPMN diagrams to be interpreted without ambiguity. BPMN, being a graphical language, lends itself to being used by domain specialists without computing expertise.

This paper has two main contributions. Firstly, we introduce a generic observation workflow model *OWorkflow*, an extension of the workflow model implemented in the CancerGrid trial model [5], customised for modelling empirical studies declaratively. Secondly, we describe bidirectional transformation functions between *OWorkflow* and a subset of BPMN. While the transformation from BPMN to *OWorkflow* provides a medium for empirical studies to be specified graphically as workflows, transforming *OWorkflow* to BPMN allows graphical visualisation. Moreover, the BPMN descriptions of empirical studies may be translated into BPEL processes, whereby manual and automated observations may be simulated and executed respectively, and both of which can be monitored during the enactment of studies. Furthermore, BPMN has a formal semantics and the transformation induces such behavioural semantics to *OWorkflow*. This means empirical study plans can now be formally specified, and interpreted without ambiguity.

The rest of this paper is structured as follows. Section 2 describes the abstract syntax and the semantics of our workflow model *OWorkflow*. Here we only describe the semantics informally, even though a formal semantics has been defined via transformation to BPMN. Section 3 describes the syntax and the semantics of a subset of the BPMN; the complete definition of its formal semantics may be found in our other paper [16]. Section 4 details the bidirectional transformation function between *OWorkflow* and the subset of BPMN by introducing BPMN constructs that are used as building blocks for modelling *OWorkflow*. We have implemented both the syntax of our observational workflow model and BPMN and the transformation functions in the functional programming language Haskell [1]. Section 5 discusses how this transformation allows simulation and automation of empirical studies, and how formalisation has assisted the transformation process. Section 6 discusses related work and concludes this paper.

2 Abstract Syntax of Observational Workflow

In this section we describe the observation workflow model *OWorkflow*. This model generalises the clinical trial workflow model defined in the CancerGrid project [5]. Each workflow is a list of parameterised *generic* activity interdependence sequence rules, where each rule models the dependency between the prerequisite and the dependent observations. Figure 2 shows the abstract syntax of *OWorkflow*. Each sequence rule is implemented using the Haskell tuple type `EventSequencing`, which contains a single constructor `Event` and each observational workflow hence is a collection of sequence rules.

```
type OWorkflow = [EventSequencing]
data EventSequencing =
  Event ActId PreAct Condition Condition Obv [RepeatExp] Works

data ActId = START | STOP | NORMAL_STOP | ABNORMAL_STOP |
             OtherId String
```

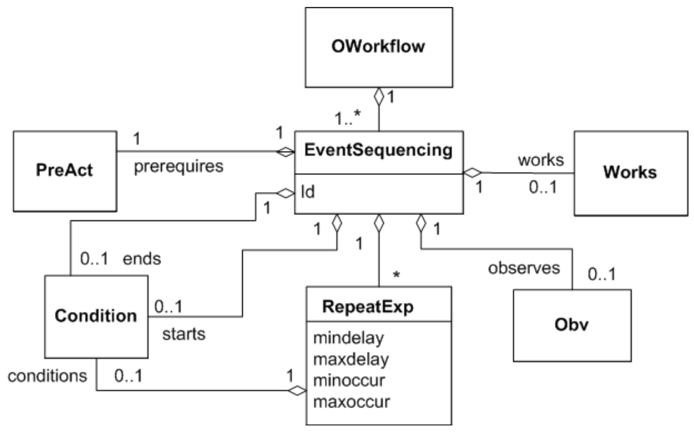


Figure 2: Abstract syntax of *OWorkflow*

Each sequence rule is identified by a unique name of type `ActId` from the first argument of the constructor `Event`, and contains zero or more dependent observations. There are four reserved names of type `ActId` for identifying a start, a generic termination, a successful termination and an unsuccessful termination of a workflow execution. Each rule defines a structural composition of dependent

[1] `http://www.haskell.org`

observations of type `Maybe Obv`, in the fifth argument of the sequence rule. (A value of type `Maybe a` either contains a value of type `a`, or is empty.)

```
data Obv = ChoiceD [Obv] | ParD [Obv] | SeqD [Obv] | Da Act
type Act = (ActId,Duration,Duration,Condition,ActType)
```

We define a single dependent observation by the tuple type `Act`, whose first component is a unique name from a set of names `ActId` distinct from those which identify sequence rules. When performing dependent observations specified by each sequence rule, there exists a delay: a range with a minimum and a maximum duration, specified by the second and third component of `Act` of type `Duration`.

```
data Duration = UNBOUNDED | Dur String
```

Each duration records a string value in accordance with XML schema datatypes [17]. For example in a clinical trial, the follow-up observation should be made between two and three months after all observations associated with the end of the treatment have been carried out. Each observation may either be a manual or an automated observation, denoted by the fifth component `ActType` of `Act`.

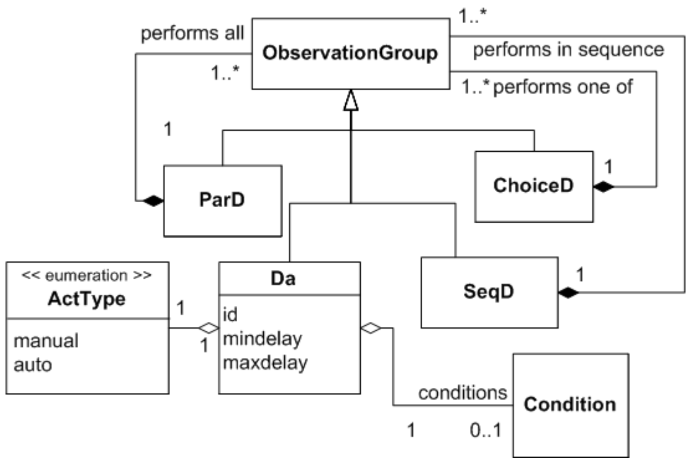


Figure 3: Abstract syntax of an *observation group*

Each composition of observations defines an *observation group*, as shown in Figure 2. Figure 3 shows the abstract syntax of an *observation group*. Each observation group structurally conforms to Kiepuszewski's *structure workflow model* [6, Section 4.1.3]. The following inductive definition of compositional rules of an observation group follows from the definition of `Obv`:

1. If `obv :: Act` is a single observation, then `Da obv :: Obv` defines an observation group and is structurally conforms the structure workflow model, and it yields to completion when the observation identified by `obv` has been made.

2. Let `obv1,...,obvN :: Obv` be observation groups; their sequential composition `SeqD [obv1,...,obvN] :: Obv` also defines an observation group

and it structurally conforms the structure workflow model. Given an observation group `SeqD obvs :: Obv`, we describe its semantics inductively:

(a) The initial observation group is `head obvs`, its initial observation may be performed when all observations associated with `SeqD obvs`'s prerequisite rules have been made;

(b) For any observation group `obvs!!n` where n ranges over `[1..(length obvs - 1)]`, its observation may be performed when observations from the group `obv!!(n-1)` have been made;

(c) `SeqD obvs` yields to completion when observations from the group `last obvs` have been made.

3. Let `obv1,...,obvN :: Obv` be n observations groups, an application of choice over them `ChoiceD [obv1,...,obvN] :: Obv` also defines an observation group, it structurally conforms the structure workflow model, and it yields to completion when observations from one of the observation groups from the given list have been made;

4. Let `obv1,...,obvN :: Obv` be n observations groups, their parallel composition `ParD [obv1,...,obvN] :: Obv` also defines an observation group and it structurally conforms the structure workflow model, the observation from each of observation groups from the given list may be interleaved, and the group yields to completion when observations from all of the observation groups from the given list have been made;

5. Nothing else defines an observation group.

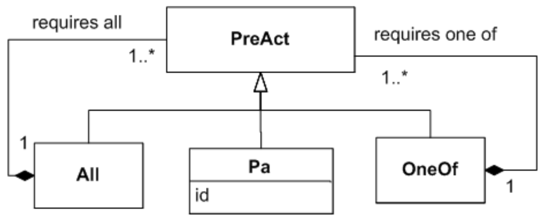


Figure 4: Abstract syntax of a *prerequisite rule group*

Dependent observations are performed after the observations associated with the *prerequisite* sequence rules, identified by the data type `PreAct`, are completed. For example in a clinical trial the follow-up observation should be made after all observations associated with the end of the treatment have been carried out. A prerequisite is a collection of names that identifies preceding sequence rules, recorded in the second argument of `Event`. It is defined using the data type `PreAct`; we call each collection a *prerequisite rule group*. Figure 4 shows the abstract syntax of a *prerequisite rule group*.

```
data PreAct = All [PreAct] | OneOf [PreAct] | Pa ActId
```

The constructor Pa defines a single prerequisite rule by its argument, which yields to completion when all observations associated with the rule identified by the argument are made. The branching constructor All denotes synchronisation over its given list of prerequisite rule groups; this yields to completion when observations from all of the prerequisite rules groups from the given list have been made. The branching constructor OneOf denotes an exclusive merge over its given list of prerequisite rules groups; this yields to completion when observations from one of the prerequisite rules groups from the given list have been made. Inductively we describe the compositional rules of a prerequisite rule group:

1. If `id :: ActId` is a unique name that identifies a particular sequence rule, then `Pa id :: PreAct` defines a prerequisite rule group;
2. If `pa1,...,paN :: PreAct` are n prerequisite rule groups, then the synchronisation of them `All [pa1,...,paN] :: PreAct` also defines a prerequisite rule group;
3. If `pa1,...,paN :: PreAct` are n prerequisite rule groups, then their exclusive merge `OneOf [pa1,...,paN] :: PreAct` also defines a prerequisite rule group;
4. Nothing else defines a prerequisite rule group

Each sequence rule also defines a list, possibly empty, of *repeat* clauses described by the sixth argument, typed [RepeatEx], of Event. Each clause specifies the condition, the minimum and the maximum numbers of iterations and the delay between iterations for the dependent observations of the sequence rule. These clauses are evaluated sequentially over the list after one default iteration of performing the rule's dependent observations.

```
type RepeatExp = (Duration,Duration,Repeat,Repeat,Condition)
data Repeat = Rep Int | Any
```

Each clause, of type RepeatExp, contains a condition specified by the fifth component of type Condition. Our definition of Condition extends the *skip logic* used in the CancerGrid Workflow Model [5]. Specifically, its syntax captures expressions in conjunctive normal form.

```
data Condition = None | Nondeter | And [Alter]
data Alter = Alt [SCondition]
type SCondition = (Range,Property)
data Range = Bound RangeBound RangeBound | Emu [String]
data RangeBound = Abdate Duration | Abdec Float | Abint Int |
                  Rldate Property Duration | Rldec Property Float |
                  Rlint Property Int
```

Each condition `c :: Condition` yields a boolean value and is either empty (true), denoted by the nullary constructor None, nondeterministic denoted by the nullary constructor Nondeter, or defined as the conjunction of clauses, each of which is a disjunction of boolean conditions, of type SCondition. The type SCondition is satisfied if the value of specified property (typed Property) falls into the specified range (typed Range) at the time of evaluation. The specified property is a name that identifies a particular property in the domain of the empirical study and this corresponds the local property to the whole BPMN process [9, Section 8.6.1]. Note while our formal semantics of BPMN [16] allows behavioural process-based specifications and corresponding verifications for *OWorkflow*, it is at a level of abstraction in which we do not directly model the value of each properties.

The range may be an enumeration of values via the constructor `Emu`, or a closed interval of two numeric values via the constructor `Range` over two arguments of type `RangeBound`, which may be absolute or relative to a property.

Given a list of repeat clauses `res :: [RepeatExp]` defined in some sequence rule `sr`, we may inductively define the evaluation of the repeat clauses:

1. The initial repeat clause is `head res`. It is evaluated after the default iteration of `sr`'s dependent observations have completed, the condition defined by the fifth component of `head res` is satisfied;

2. A given repeat clause terminates if either its maximum number of repetitions has been reached, or its minimum number of repetitions has been reached and the condition is no longer satisfied;

3. For any clause `res!!n` where n ranges over `[1..(length res - 1)]`, it may be evaluated after the evaluation of the clause `res!!(n-1)` terminates;

4. `res` terminates when `last res` terminates.

For example, the follow up sequence rule of a clinical trial might specify that follow up observations should be made every three months for three times after the default observations have been made, after which observations should be performed every six months for four times.

Each sequence rule might also include work units, recorded by the last argument of the constructor `Event`. Each work unit represents an empirical procedure such as administering a medical treatment on a patient in a clinical trial. In each sequence rule, the procedure defined by work units are interleaved with the rule's observations. Each collection of work units is defined by the data type `Works` and is called *work group*.

```
data Works = ChoiceW [Works] | ParW [Works] | SeqW [Works] | Wk Work
```

The type `Work` records a unique name that identifies a particular empirical procedure. Our definition of work group also structurally conforms to Kiepuszewski's structure workflow model, and both its abstract syntax and compositional rules are similar to those of observation groups.

Finally the third and fourth arguments of a sequence rule are two conditional statements, each of type `Condition`. While the third argument defines the condition for enacting the sequence rule, the fourth argument defines the condition for interrupting the enactment of the sequence rule.

3 Abstract Syntax of BPMN

In this Section we describe the syntax of our chosen subset of BPMN and informally their semantics. For the purpose of specifying and simulating observa-

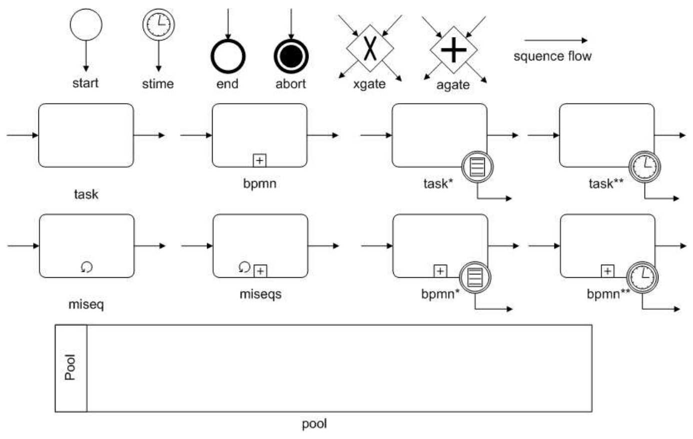


Figure 5: States of BPMN diagram

tional workflow `OWorkflow`, our implementation of BPMN states captures only a subset of the abstract syntax of BPMN defined in our other paper [16].

States in our subset of BPMN [11] can either be tasks, subprocesses, multiple instances or control gateways, each linked by a normal sequence or an exception sequence flow. A normal sequence flow can be either incoming to or outgoing from a state and have associated guards; an exception sequence flow, depicted by the state labelled *task**, *bpmn**, *task*** and *bpmn***, represents an occurrence of error within the state. A sequence of flows represents a specific control flow instance of the business process. Figure 6 shows the abstract syntax of a BPMN state, we describe each state using a Haskell data type `State`

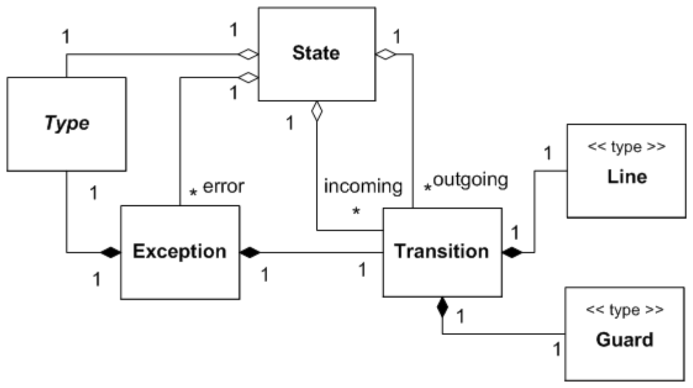


Figure 6: Abstract syntax of BPMN state

```
data State = State Type [Transit] [Transit] [(Type,Transit)]
```

where its first argument, of type `Type`, records the type of state; second and third arguments, both of type `[Transit]`, records a list of incoming and outgoing transitions respectively. We use the type `Transit` to record each normal sequence and exception sequence flow, and it holds a pair of a boolean guard `Guard` and a unique name, of type `Line`, that identifies that a line.

```
type Transition = (Guard,Line)
```

The fourth argument, of type `[(Type,Transit)]`, records a list of pairs where each pair records a particular type of exception and its outgoing transition.

We record each type of states by the type `Type`

```
data Type = Itime Time | Stime Time | Irule BCondition |
            Agate | Xgate | Start | End Int | Abort Int |
            Task TaskName TaskType| Bpmn BName BpmnType |
            Miseq TaskName Int TaskType BCondition |
            Miseqs BName Int BpmnType BCondition
```

where each type of state is presented graphically in Figure 5. In the figure, there are two types of start states *start* and *stime*. A *start* state models the start of the business process in the current scope by initiating its outgoing transition. It has no incoming transition and only one outgoing transition. Its type is implemented by the nullary constructor `Start`. The *stime* state is a variant start state and it initiates its outgoing transition when a specified duration has elapsed. This type of state is implemented by the constructor function `Stime` and it takes an argument the duration of type `Time`,

```
data Time = NOBOUND | MkTime Direction Int Int Int Int Int Int
data Direction = Pve | Nve
```

where `NOBOUND` denotes an unspecified duration and the constructor `MkTime` records a specific duration by its arguments, and it represents six-dimensional space of the XML schema data type *duration* [17].

There are two types of end states *end* and *abort*. An *end* state models the successful termination of an instance of the business process in the current scope by initialisation of its incoming transition. It has only one incoming transition with no outgoing transition. Its type is implemented by the constructor `End` which takes a unique integer for identifying a particular end state. The *abort* state is a variant end state and its models an unsuccessful termination, usually an error of an instance of the business process in the current scope. Its type is implemented by the constructor `Abort` which takes a unique integer for identifying a particular abort state.

Our subset of BPMN contains two types of decision state, *xgate* and *agate*. Each of them has one or more incoming sequence flows and one or more outgoing sequence flows. Their state types have been implemented by Haskell type nullary constructors `Xgate` and `Agate` respectively. An *xgate* state is an exclusive gateway, accepting one of its incoming flows and taking one of its outgoing flows; the semantics of this gateway type can be described as an exclusive choice and a simple merge. An *agate* state is a parallel gateway, which waits for all of its incoming flows before initialising all of its outgoing flows.

A *task* state describes an atomic activity and it has a exactly one incoming and one outgoing transitions. Its type is implemented by the Haskell constructor `Task` and it takes two arguments recording a unique name for identifying the activity and the task type for differentiating states that describes work units from the rest. Note it is possible to introduce this property as BPMN is an extensible notation [11, Section 7.1.3].

```
data TaskType = StandardT | WorkT
```

A *bpmn* state describes a subprocess state. it is a business process by itself and so it models a flow of BPMN states. Figure 5 depicts a collapsed subprocess state where all internal details are hidden. this state has a exactly one incoming and one outgoing transitions. Its type is implemented by the constructor `Bpmn` and it takes two arguments recording a unique name for identifying a particular subprocess and the subprocess type `BpmnType` for differentiating subprocess states modelling different parts of a sequence rules.

```
data BpmnType = SequenceB | ScopeB | DependentB | WorkB | RepeatB
```

Also in Figure 5 there are graphical notations labelled *task**, *bpmn**, *task*** and *bpmn***, which depict a task state and a subprocess state with an exception sequence flow, . As mentioned above, we describe exception sequence flows of a state by the fourth argument of the constructor `MkState`. There are two types of exception associated with task and subprocess states in our subset of BPMN states. Both states *task** and *bpmn** are examples of states with a *conditional* exception flow, we implement this type of exception flows by the constructor `Irule`, which takes an argument specifying the condition to interrupt the execution of the state. The states *task*** and *bpmn***, on the other hand, are examples of states with a exception flow of an *expiration*, we implement this type of exception flows by the constructor `Itime`, which takes an argument specifying the duration until expiration.

Each task and subprocess may also be defined as *multiple instances.* The *miseq* state type represents serial multiple instances, where the specified task is repeated in sequence. This has been implemented by the Haskell type constructor `Miseq`, and it takes four arguments, recording a unique name that identifies the task to be repeated, the maximum number of repetition of the specified task, the task type and condition to be evaluated before the task begins. We have implemented the type `BCondition` to record a conditional statement,

```
data BCondition = And [Clause] | SgB Clause | NoCond
data Clauses = Or [Literal] | SgC Literal
data Literal = Gt Quantity Quantity | Lt Quantity Quantity |
               El Quantity Quantity | Ne Quantity Quantity
```

where each statement is a propositional statement in conjunctive normal form. Each data value of type `BCondition` may be defined using either the constructor `And`, which defines a conjunction over a list of *clauses*, each of type `Clause`, the constructor `SgB`, which defines a single clause, or the nullary constructor `NoCond`, which defines no condition and is a tautology. The type `Clause` is defined using either the constructor `And`, which defines a disjunction of *literals*, each of type `Literal` or the constructor `SgC`, which defines a single literal. The type `Literal` defines binary numerical equalities and inequalities using constructors `Gt`, `Lt`, `El` and `Ne` for greater than, less than, equal and not equal respectively over a pair of numerical quantities, each of type `Quantity`, of the same numerical type.

```
data Quantity = Pty Property | Rge Range
data Property = Nm String VType
data VType = EmT | InT | FlT | TiT
data Range = Emv [String] | Inv Int Int | Flv Float Float |
             Tiv Time Time
```

Each quantity defines either a unique name that identifies a property of type `Property` associated with a particular state, a closed interval of numerals and durations or an enumeration of values. The type *miseqs* is the subprocess counterpart of *miseq*. Its type is implemented by the constructor `Miseqs`.

The graphical notation *pool* in Figure 5 forms the outermost container for a single business processes; only one process instance is allowed at any one time.

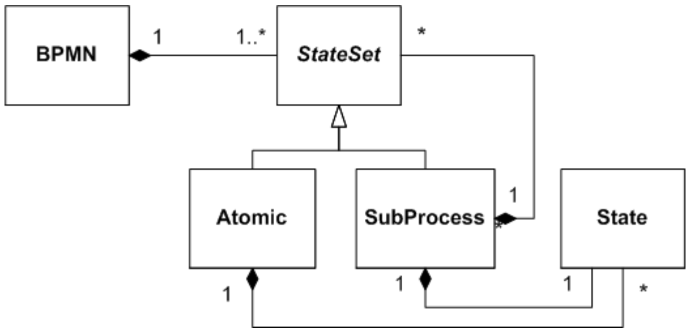


Figure 7: Abstract syntax of BPMN diagram

Figure 7 shows the abstract syntax of a BPMN diagram, where each diagram is a collection of `StateSet`. Below is the corresponding data type `StateSet` in Haskell.

```
data StateSet = Atomic [State] | SubProcess State [StateSet]
type BPMN = [(CName,[StateSet])]
```

Each `StateSet` defines either a list of non-subprocess states by the constructor `Atomic`, or a subprocess state by the constructor `SubProcess`, which records the type and sequence flows of the subprocess states by its first argument, and the subprocess's constituent states by its second arguments, of type `[StateSet]`. A BPMN diagram hence is a list of pairs, each records the states of a single local composition diagram, typed `[StateSet]`, and the name that identifies it. We assume states of each BPMN diagram are *well-formed* [16], and introduce the notion of *well-formedness* to the representation of BPMN diagrams to facilitate our transformation process by the following definition

Definition 1 *Given a representation of some BPMN diagram `bp`, it is **well-formed** if for each local composition diagram `comp` in `[ lc | (n,lc) <- bp ]`, its non-subprocess states are recorded in `head comp` and are well-formed, and all its subprocess states are recorded in `tail comp` and they are also well-formed.*

We hence identify each BPMN diagram within a collaboration diagram [16] by a unique name, of type `CName` and we therefore defines a BPMN diagram using the type `BPMN`, which is a list of pairs of name and its corresponding BPMN diagram.

4 Transformation

In this section we describe the bidirectional transformation between observation workflows of type `OWorkflow` and their corresponding subset of BPMN diagrams. Specifically we have implemented a total function transforming `OWorkflow` to `BPMN` and its inverse, a partial function transforming a subset of `BPMN` to `OWorkflow`.

```
w2b :: OWorkflow -> BPMN
b2w :: BPMN -> OWorkflow
```

Here we explain the functions informally. We initially describe transformation between a single sequence rule to its corresponding BPMN subprocess state by explaining the transformation over each of the components that makes up the 7-tuple of a sequence rule. We describe the transformation of individual components by introducing some building blocks in BPMN, which may be mapped to those components. We then describe the notion of *scoping*, the transformation rules of Prerequisite, and composition of sequence rules to model a complete observation workflow.

4.1 Observation

Figure 8 shows an *expanded* BPMN subprocess state depicting a single dependent observation, of type `Act`. According to the behavioural semantics of an observation, an observation may be performed after a delay ranging from the minimum to the maximum duration, provided that its associated condition is satisfied. The delay range is graphically modelled by first modelling minimum duration as the *stime* state (timer start event), and then modelling the duration ranges from the elapse of the minimum duration to the maximum duration using a task state which halts for an unknown duration, with an expiration exception flow, of which the expiry duration is the difference between maximum and minimum durations of the delay. We use a *xgate* (exclusive choice) decision gateway state for accepting either the task state's outgoing transition or its expiration exception flow.

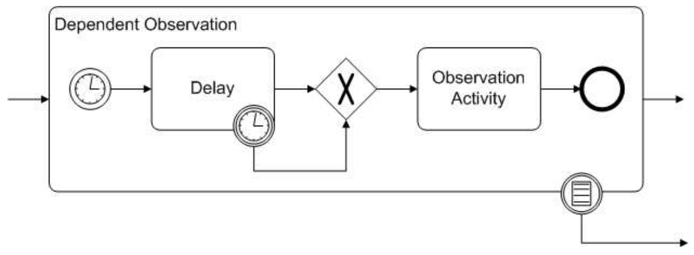


Figure 8: A BPMN subprocess state depicting a single observation.

The decision gateway is then followed by a task state, which models the actual observation itself and is identified by applying the function `idToTName` to the identifier of the observation being mapped.

```
idToTName :: ActId -> TaskName
```

An *end* state follows immediately for terminating the execution of the subprocess. The subprocess itself has one incoming and one outgoing transition, denoted as the *outermost* incoming and outgoing transitions respectively. We have implemented the function `mkAct` to automate the transformation from the subprocess state modelling an observation of type `Obv`, encapsulating a single observation of type `Act` via the constructor function `Da`.

```
mkAct :: StateSet -> Obv
mkAct (SubProcess s st) = Da ((getDNme st),(getMaxT st),(getMinT st),
                              (getCond (SubProcess s st)))
```

where the function `getDNme` takes a list of states of a subprocess state describing a dependent observation and returns. the identifier of that observation; functions `getMaxT` and `getMinT` each takes the same argument as `getDNme` and returns the maximum and minimum delay, of type `Duration` respectively. The function `getCond` takes the subprocess state, together with its constituent states and returns a *skip logic* condition [5].

We have also implemented the function `mkDpt` to transform a single observation, of type `Obv` to a BPMN subprocess modelling that observation,

```
mkDpt :: Act -> Line -> ([StateSet],Line)
mkDpt (id,min,max,cond) ln =
      ([ (SubProcess (MkState (Bpmn (idToBName id) SequenceB)
           [(True,ln)] [(True,nl)] (mkExp cond) 0) sts) ], nl)
      where (sts,nl) = mkdIntern (id,min,max) (inLine ln)
```

where the function `idToBName` maps the name of an observation to a unique name for identifying the subprocess, the function `mkExp` maps a condition to a set of exception flows and the function `mkdIntern` maps the three components of an observation to its corresponding subprocess's constituent states. Note the function `mkDpt` also takes a fresh line name of type `Line` for defining the transitions of the subprocess states and returns another fresh line name for constructing other parts of the diagram. The function `inLine` takes a used line name and returns a fresh one, and the default guard for both BPMN normal and exception sequence flow is `True`.

4.2 Groups

Each sequence rule contains zero or more observations and work units. Whereas the transformation of a single observation has been described in Section 4.1, each work unit is modelled as a task state, of which the name that identifies the task is obtained by applying the function `workToTask` on the unique identifier of the work unit. Conversely, the function `taskToWork` is defined to map a task state name to the unique name of the work unit it models. One or more observations compose into an observation group, which has been defined inductively in Section 2. Similarly one or more work units compose into a work group. Due to the conformity of both types of compositions to the structured workflow model [6] as mentioned in Section 2, we have generalised the notion of *group* and here we describe the transformation between a group and its corresponding BPMN subprocess state, which may be applied to both observation group and work group.

Semantically a group describes the control flow of a collection of activities, Figure 9 shows a BPMN subprocess state describing an observation group, which is a collection of observations. In the figure each individual observation is modelled by a *collapsed* subprocess state to maintain the hierarchical structure of

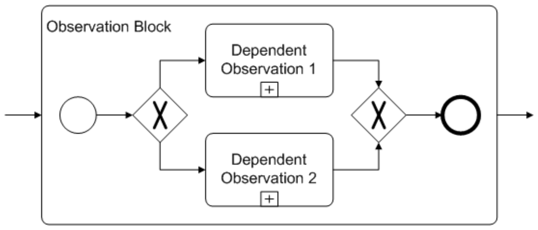


Figure 9: A BPMN subprocess state depicting an observation group.

the BPMN diagram. Specifically the figure depicts a BPMN subprocess state describing an observation group defined by the constructor `ParD` over a list of two observations, each by the constructor `Da`. Figure 10, on the other hand, shows a BPMN subprocess state describing a work group, specifically the figure depicts a BPMN subprocess state describing a work group defined by the constructor `ParW` over a list of two work units, each by the constructor `Wk`. We

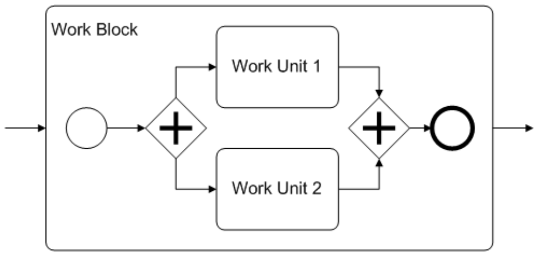


Figure 10: A BPMN subprocess state depicting a work group.

describe informally the transformation rules for a group as follows:

1. Given group *go* defined by the constructor over a single activity *sa*, specifically `Da` applied over a single observation for an observation group and `Wk` applied over a single work unit for a work group respectively, we transform *sa* according the type of the activity, for an observation, the transformation rule has been described in Section 4.1, and a work unit is simply represented by a task state, of which the task name is defined by applying `workToTask` to the name of the work unit. We use *sa*'s outermost incoming and outgoing transitions as *go*'s outermost incoming and outgoing transitions.

2. Given a group *go* defined by some parallel constructor over a list of n groups, specifically `ParD` and `ParW` applied over a list of observation groups and work groups respectively, where $n \geq 1$, the corresponding BPMN states are two *agate* decision gateways. One of which has one incoming transition, denoted as the *go*'s outermost incoming transition, and n outgoing transitions, each matching the outermost incoming transition from one of the n groups, and the other one has n incoming transitions, each matching the outermost outgoing transition from one of the n groups, and

one outgoing transitions, denoted as the *go*'s outermost outgoing transition. The transformation of the n groups are defined recursively.

3. Given group *go* defined by some choice constructor over a list of n groups, specifically `ChoiceD` and `ChoiceW` applied over a list of observation groups and work groups respectively, where $n \geq 1$, the corresponding BPMN states are two *xgate* decision gateways where their transitions are defined similarly to the above rule, and the transformation of the n groups are defined recursively.

4. Given an observation group *go* defined by the sequential constructor over a list of n groups, specifically `SeqD` and `SeqW` over a list of observation groups and work groups respectively, where $n \geq 1$, the outermost outgoing transition of each group is matched by the outermost incoming transition of its next group. The outermost incoming transition of the first group defines the outermost incoming transition of *go*, and the outermost outgoing transition of the last group defines the outermost outgoing transition of *go*.

We have implemented the function `getObv` to transform the subprocess state describing an observation group to an observation group of type `Obv`.

```
getObv :: StateSet -> Obv
getInv :: StateSet -> Works
```

Similarly, we have implemented the function `getInv` to transform a work group of type `Works`. Conversely, we have implemented the function `extObv` to transform an observation group of type `Obv` to a subprocess state describing that group,

```
extObv :: Line -> Obv -> ([StateSet],Line)
extObv ln (ChoiceD dpts) = extDptM Xgate ln dpts
extObv ln (ParD dpts) = extDptM Agate ln dpts
extObv ln (Da act) = if (elem (fst4 act) specialId)
                     then ([],ln) else mkDpt act ln
extObv ln (SeqD dpts) = ((concat.fst) seqs,(last.snd) seqs)
  where seqs = (unzip . extDptS ln) dpts
```

where the function `extDptM` takes a list of observation groups and constructs the required decision gateways and recursively transforms a list of observation groups according to Rules 2 and 3 described above; the function `mkDpt` has been described in Section 4.1 and the function `extDptS` recursively transform a list of observation groups taken from the argument of the constructor `SeqD` according to rule 4. A corresponding function has been implemented for work groups.

4.3 Repeat Clauses

This section describes informally the transformation between a list of repeat clauses, each of type `RepeatExp`, and its corresponding BPMN subprocess state. Figure 11 shows a BPMN subprocess modelling a single repeat clause. According to the semantics of a repeat clause, each repeat clause in a sequence rule repeats all dependent observations defined in that rule; the number of repetitions from each clause ranges between a minimum and a maximum value, and there is a delay, ranging between a minimum and a maximum duration, before each repetition can start. We model the delay range of a repeat clause graphically according to the transformation rules defined for a single observation in Section 4.1.

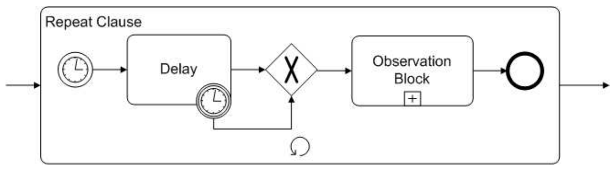


Figure 11: A BPMN subprocess state depicting a repeat clause.

We model each repeated observations as a subprocess state according the transformation of groups in Section 4.2. An *end* state follows immediately for terminating the execution of the subprocess. The subprocess, which defines the repeat clause, is a multiple instance *miseqs* state, and it has one incoming and one outgoing transition, denoted as the *outermost* incoming and outgoing transitions respectively. The multiple instance subprocess state is implemented by the Haskell type `Miseqs` which takes an integer value to specify the maximum number of repetitions and a condition to specify the conjunction of the minimum number of repetitions required and the clause's conditional statement. Below shows an example of how to model the clause's repetition range and conditional statement,

```
rep01 = Miseqs "Treatment" 2 RepeatB (And [cond01,cond02])
con01 = Sgl (Lt (Pty (Nm "LoopCounter" InT)) (Rge (Inv 5 10)))
con02 = Sgl (Eq (Pty (Nm "Abnormal Blood Count" EmT))
              (Rge (Emv ["VHigh","VLow"])))
```

where the state type `rep01` records information about a repetitive observation during a study of an effect of a medical intervention. It contains a conjunction of two conditions. Condition `cond01`, is satisfied if the current number of repetition is less than a value *randomly* chosen over the closed interval `[5..10]`, and Condition `cond02` is satisfied if the abnormal blood count of the patient is either very high or very low.

A list of repeat clauses is therefore transformed iteratively over each clause starting from head of the list, similar to the transformation of a group for some sequential constructor described in the Rule 4 in Section 4.2. Figure 12 shows a BPMN subprocess state representing a list of two repeat clauses. Individual repeat clause is shown as collapsed subprocess state.

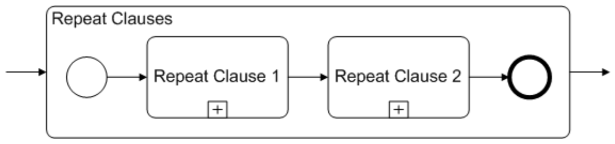


Figure 12: A BPMN subprocess state depicting a list of two repeat clauses.

4.4 Sequence Rules

We have so far described the transformation rules for repeat clauses, observation and work groups, constituting the fifth, sixth and seventh elements of a sequence rule's 7-tuple. In this section we apply these transformation rules and detail the transformation of a complete sequence rule. Figure 13 shows a

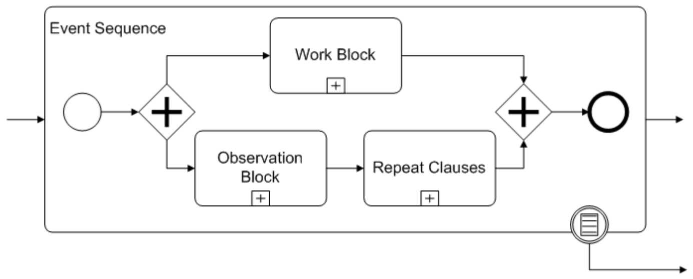


Figure 13: A BPMN subprocess representing a single sequence rule.

BPMN subprocess state representing a single sequence rule. The subprocess state is defined by three other subprocess states, collapsed in the figure, which model observations, work units and repeat clauses defined in the sequence rule. A sequence rule is enacted by first performing all its observations once, modelled by the subprocess *observation block*, after which the list of repeat clauses, modelled by the subprocess state *repeat clauses* is evaluated. As explained in Section 2, work units are empirical procedures and their executions are interleaved with their corresponding observations, hence we use an *agate* decision gateway state to initialise both observations and work units. We do not constrain how work units are interleaved with observations as our current workflow model focuses on the specification of observations, therefore it solely depends on the study planners. Note if no work unit is defined in the sequence rule, the corresponding subprocess will not have *agate* states and will be represented by a sequential composition of the *observation block* and *repeat clauses* states.

Finally we associate a *conditional* exception sequence flow with each subprocess state to model the enacting and the interrupting conditions of the sequence rule. where the type `Irule` takes a disjunction, translated into conjunctive normal form, of the two conditions, where the enacting condition is conjoined with the atomic proposition *status* == *ready*, signifying the state is ready to be enacted, and the interrupting condition is conjoined with the proposition *status* == *completing*, signifying the state has been enacted. For example, in a clinical trial a sequence rule may be defined for administering a new drug on a patient, where insulin level of the patient is to be monitored before and during the treatment. The following two conditions might have been specified,

```
start = Ands [Ors [(Emu ["Normal"],"Insulin Levels")]]
terminating = Ands [Ors [(Emu ["Low"],"Insulin Levels")]]
```

where the sequence rule may be carried out if `enact` is satisfied and the enactment must be aborted if `interrupt` is satisfied.

The following shows the translation these conditions into appropriate conjunctive normal form, where the statement `condS` is satisfied either both the state is ready to be enacted and insulin level is not normal or both the state is being enacted and insulin level is low.

```
cA = El (Pty (Nm "this.status" EmT)) (Rge (Emv ["Ready"]))
cB = Ne (Pty (Nm "Insulin Level" EmT)) (Rge (Emv ["Normal"]))
cC = El (Pty (Nm "this.status" EmT)) (Rge (Emv ["Completing"]))
cD = El (Pty (Nm "Insulin Level" EmT)) (Rge (Emv ["Low"]))
condS = And [Or [cA,cC],Or [cA,cD],Or [cB,cC],Or [cB,cD]]
```

Both `cA` and `cC` are propositions of the state's status, and both `cB` and `cB` are proposition translated from the two conditional statements above.

4.5 Scopes and Prerequisites

An observation workflow, of type `OWorkflow`, is a list of sequence rules connected according to each rule's prerequisite. Figure 14 shows a single observation workflow modelled as a BPMN diagram. It consists three sequence rules and one

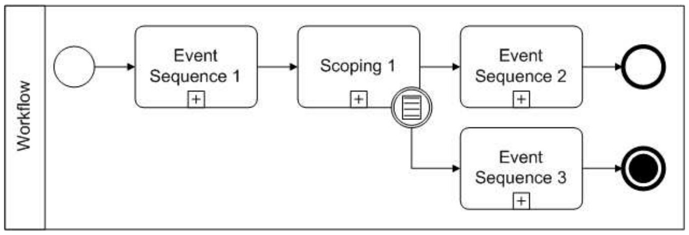


Figure 14: A BPMN diagram describing a single observation workflow.

collapsed *scoping* subprocess state. The expanded version of the scoping subprocess state is shown in Figure 15, where there are two subprocess states, each

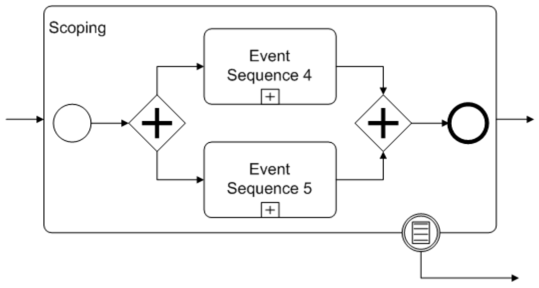


Figure 15: A BPMN scoping subprocess state.

modelling a sequence rule. Scoping is an extension of prerequisite and it ensures correct dependencies can be constructed between sequence rules.

Prerequisite is a collection of names that identifies preceding sequence rules, and is recorded in the second element of each sequence rule. Here we describe the translation rules to build dependencies between sequence rules according to their prerequisite. First we describe the derivation, the application and the associated functions of a *sequence paths* list.

A sequence paths list, of type `[[ActId]]`, contains a list of syntactic path from the start of the workflow enactment to a termination via one or more sequence rules, each identified by its rule name. We have implemented the function `preList` to derive the sequence paths list.

```
preList :: [EventSequencing] -> [ActId] -> PreAct -> [[ActId]]
preList es ans (Pa START) = [[START]++ans]
preList es ans (OneOf pas) = concatMap (preList es ans) pas
preList es ans (All pas) = concatMap (preList es ans) pas
preList es ans (Pa a) =
        preList es ([a]++ans) ((getPr.head) (filter (equalNm a) es))
```

For example consider the following simple observation workflow,

```
[Event (Id "SEQ1") (Pa START),
 Event (Id "SEQ2") (Pa (Id "SEQ1")),
 Event (Id "SEQ3") (Pa (Id "SEQ1")),
 Event (Id "SEQ4") (Pa (Id "SEQ2")),
 Event (Id "SEQ5") (Pa (Id "SEQ2")),
 Event (Id "SEQ6") (OneOf [Pa (Id "SEQ4"),Pa (Id "SEQ5")]),
 Event (Id "SEQ7") (All [Pa (Id "SEQ3"),Pa (Id "SEQ6")]),
 Event NORMAL_STOP (Pa (Id "SEQ7"))]
```

where we have only shown each rule's name and its prerequisite, its corresponding sequence paths list are derived as follows with three sequence paths leading from the start to a termination of the workflow.

```
[[START,Id "SEQ1",Id "SEQ3",Id "SEQ7"],
 [START,Id "SEQ1",Id "SEQ2",Id "SEQ4",Id "SEQ6",Id "SEQ7"],
 [START,Id "SEQ1",Id "SEQ2",Id "SEQ5",Id "SEQ6",Id "SEQ7"]]
```

Each sequence rule is connected to its preceding rules via prerequisite, we begin constructing the dependencies by recursively evaluating the prerequisite of each rule starting from the rule, of which the identifier immediately precedes termination in the sequence paths list, in the above example the starting rule is identified by the name `SEQ7`. The evaluation follows sequence paths list until it reaches the beginning of the list, the reserved name `START`, which identifies the start of the workflow.

When evaluating a prerequisite defined by one of the branching constructors, `OneOf` or `All`, each branch is evaluated independently, therefore it is necessary to derive the *merging point* of all the branches. we have implemented the function `mPoint` over the sequence paths list to derive the merging point.

```
mPoint :: [[ActId]] -> ActId
mPoint pl = case find (com (pl \\ [shl])) (reverse shl) of
                      Just a -> a
                      Nothing -> OtherId ""
  where shl = head (filter ckl pl)
        sh = minimum (map length pl)
        ckl l = (length l) == sh
        com ps p = and (map (elem p) ps)
```

A merge point essentially is the sequence rule identified by the rule name common to all sequence paths, therefore at any point of the evaluation a sequence paths list may have only one merge point. As for the above example, the prerequisite of the rule `SEQ7` is defined by the branching constructor `All`, and the corresponding merging point is the rule identified by the name `SEQ1`.

Prerequisite may contain nested branches and when evaluating individual branch, it is also necessary to derive a sublist of a sequence paths that associates with the current path, this is because there may be different merging point for a branch to be defined by another branching constructor. Therefor we have implemented the function `subList`, which takes a list of rule names and a sequence path list, and returns a sublist of sequence paths containing those rule names from the list.

```
subList :: [ActId] -> [[ActId]] -> [[ActId]]
subList [] plist = plist
subList (i:ids) plist = subl ids (filter (elem i) plist)
 where subl id pl = if (mPoint pl /= id) then pl
                    else map (reverse.(dropWhile (/=id)).reverse) pl
```

As for the above example, while the branches defined by the prerequisite of the rule `SEQ7` has a merging point identified by the rule name `SEQ1`, the path of one of the branches contains the sequence rule `SEQ6`, which is defined by another branching constructor `OneOf`. Its corresponding sublist of sequence paths defined below and its merging point is `SEQ2`.

```
[[START,Id "SEQ1",Id "SEQ2",Id "SEQ4",Id "SEQ6"],
 [START,Id "SEQ1",Id "SEQ2",Id "SEQ5",Id "SEQ6"]]
```

Having defined the derivation and application of a sequence path list, we now describe the rules to construct dependencies. Given a list of sequence rules *SR*, the list of corresponding subprocess states *SP*, to which *SR* are transformed, and the corresponding sequence paths list *SL*, we evaluation the starting rule of *SR*, the rule which immediately precedes termination, as follows:

1. Given a rule *rl* with a prerequisite defined by the constructor `Pa` over a single rule name *id*, if *id* equals a merging point then we have reached the end of a path and returns the connected BPMN states. Otherwise we find rule *rl*2 in *SR*, identified by *id* and its corresponding subprocess state $s2$ in SP. we define the outgoing transition of the state $s2$ to be equal to the incoming transition of the *rl*'s corresponding state, we then continue to evaluate *rl*2;

2. Given a rule *rl* with a prerequisite defined by the constructor `All` over a list of rules *ids*, we define some scoping subprocess *ss*, consisting a *start* state *st*, an *end ed*, two *agate* states, $ag1$ and $ag2$, and n sequential compositions of connected states *rets*, obtained by evaluating *ids* recursively and where n is the length of *ids*. We define n incoming transitions for $ag2$ and n outgoing transitions for $ag1$. We connect each incoming transition of $ag2$ to the outgoing transition of each sequential compositions *rets*, and connect each outgoing transition of $ag1$ to the incoming transition of each sequential compositions *rets*. The outgoing transition of $ag2$ connects to *ed* and the incoming transition of $ag1$ connects to *st*;

3. Given a rule *rl* with a prerequisite defined by the constructor `OneOf` over a list of rules *ids*, we define some scoping subprocess *ss*, consisting a *start* state *st*, an *end ed*, two *xgate* states, *xg*1 and *xg*2, and *n* sequential compositions of connected states *rets*, obtained by evaluating *ids* recursively and where *n* is the length of *ids*. The definition of transitions is similar to the immediate above rule.

5 On Simulation, Automation and Formalisation

In this section we discuss briefly the application of business process management technique to empirical studies. We describe informally, via a simple example, how modelling empirical studies in BPMN allows their study plans to be simulated and partially automated by translating the BPMN diagrams into executable BPEL processes. We also discuss how modelling empirical studies in BPMN has consequently induced a formal behavioural semantics upon our observation workflow model and hence removed ambiguities in both the transformations and interpretation of *OWorkflow*.

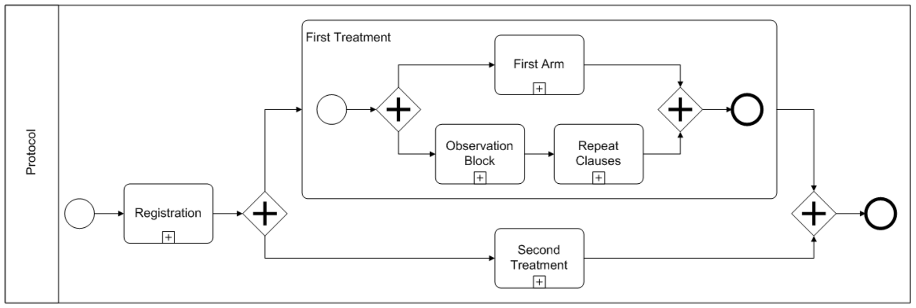


Figure 16: A BPMN diagram of a simplified clinical trial.

As useful as it is to visualise and formally specify a complete study plan, it is also beneficial to validate the plan before its execution phase, especially if the study has a long running duration, since it is undesirable to run into an error three months into the study! One method of validating a study is by simulation. When considering either simulating or automating a portion of a study, we assume the observations specified in that portion can be appropriately simulated or automated; an observation might define the action of recording a measurement from a display interfacing with a software application or submitting a web form to a web service for analysis.

Figure 16 shows a BPMN diagram of a simplified phase III chemotherapy clinical trial, which contains two sets of concurrent interventions, each interleaved with some observation consisting of submitting forms about specific medical conditions of the patients. The following shows a simplified observation group defined by the sequence rule, which is modelled by the observation block subprocess in the figure, an expanded view of the subprocess is shown in Figure 17.

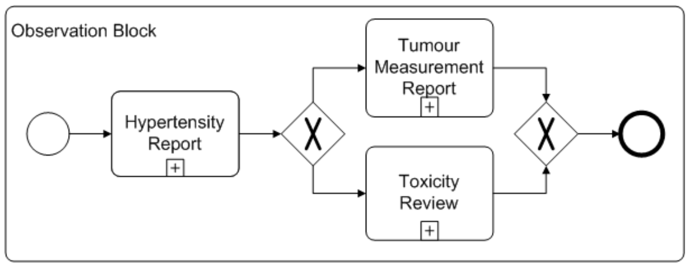


Figure 17: A BPMN diagram of an observation block.

```
(SeqD [Da (Id "Hypertensity Report", Dur "P7D",Dur "P7D",None,Manual),
 ChoiceD [Da (Id "Tumour Measurement Report", Dur "P1D",Dur "P1D",
              Ands [Ors  [(Emu ["low"],"blood pressure")]],Manual),
          Da (Id "Toxicity Review", Dur "P1D",Dur "P1D",
              Ands [Ors  [(Emu ["high"],"blood pressure")]],Manual)]])
```

While submitting a report form is a manual task, due to the transformation, it is possible to simulate this action by translating its corresponding BPMN subprocess state into the corresponding sequence of BPEL activities,

```
<sequence>
 <wait for="PT7M"><operation name="sendHypertensityReport">
   <input message="hypertensityMessage" />
 </operation></wait>
 <switch>
  <case condition="getVariableData('blood pressure') == high">
   <wait for="PT1M"><operation name="sendToxicityReview">
    <input message="toxicityMessage" />
   </operation></wait>
  </case>
  <case condition="getVariableData('blood pressure') == low">
   <wait for="PT1M"><operation name="sendTumourReport">
    <input message="tumourMessage" />
   </operation></wait>
  </case>
 </switch>
</sequence>
```

where each *wait* activity is an invocation upon the elapse of a specified duration. Since the derived BPEL process is for simulation, we scale down the specified duration of each observation. Note each invocation in a BPEL process is necessarily of a web service; if the specified observation defines an action to invoke a web service, e.g. uploading a web form, the translated BPEL operation will also be invoking that web service, and otherwise, for simulation purposes, a “dummy” web service could be used for merely receiving appropriate messages. Similarly, partial automation is also possible by translating appropriate observations into BPEL processes which may be executed during the execution phase of the study.

In recent work, a formal relative timed semantics have been given to BPMN in the process algebra CSP [16]. By defining a transformation function between

`OWorkflow` and BPMN, it has automatically induced a behavioural semantics for `OWorkflow`. In this section we briefly describe two examples. In the first

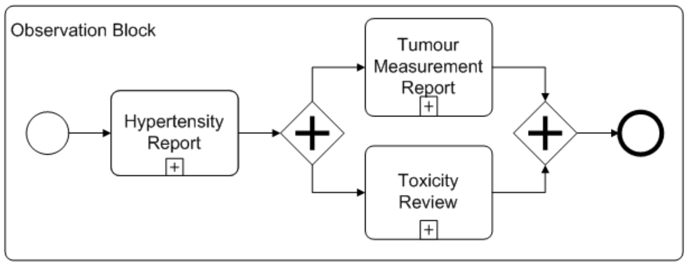


Figure 18: A BPMN diagram of a variant of observation block in Figure 17.

example, we analyse a variation of the observation block in Figure 17, shown in Figure 18. Here we show its corresponding the sequence rule.

```
(SeqD [Da (Id "Hypertensity Report", Dur "P7D",Dur "P7D",None,Manual),
 ParD [Da (Id "Tumour Measurement Report", Dur "P1D",Dur "P3D",None,Manual),
       Da (Id "Toxicity Review", Dur "P1D",Dur "P4D",None,Manual)]])
```

and also the definition of the repeat clauses.

```
[(T7D,T14D,4,4,None)]
```

We also shows the definition of the observation block and repeat clauses for the second treatment.

```
SeqD [Da (Id "Dose Report", Dur "P1D",Dur "P2D",None,Manual)]
```

```
[(T14D,T21D,3,3,None)]
```

When a clinical trial protocol is designed, one might ensure certain behavioural properties about the clinical design based on some oncological safety principles [4]. One property for this particular trial description is that

> Each dose report is to be completed before no more than 2 tumour measurement reports and toxicity reviews.

The formal semantics of BPMN in CSP [16] allows such property to be formalised as a process-based behavioural specification and allows a formal verification of the clinical trial against this specification could then done by model checking the following stable failures [14] refinement assertion

$$DR \sqsubseteq_{\sqsubseteq_F} (PLAN \setminus (\Sigma \setminus \alpha DR))$$

where DR is the process-based specification of the property in interest, $PLAN$ is the CSP process describing the semantics of the trial, in CSP we write Σ to denote the set of all possible events and αP to denote the set of possible events performed by P.

In the second example, we analyse the two different BPMN diagrams, shown in Figure 19, modelling the same observation workflow described below, omitting description of observations and work units.

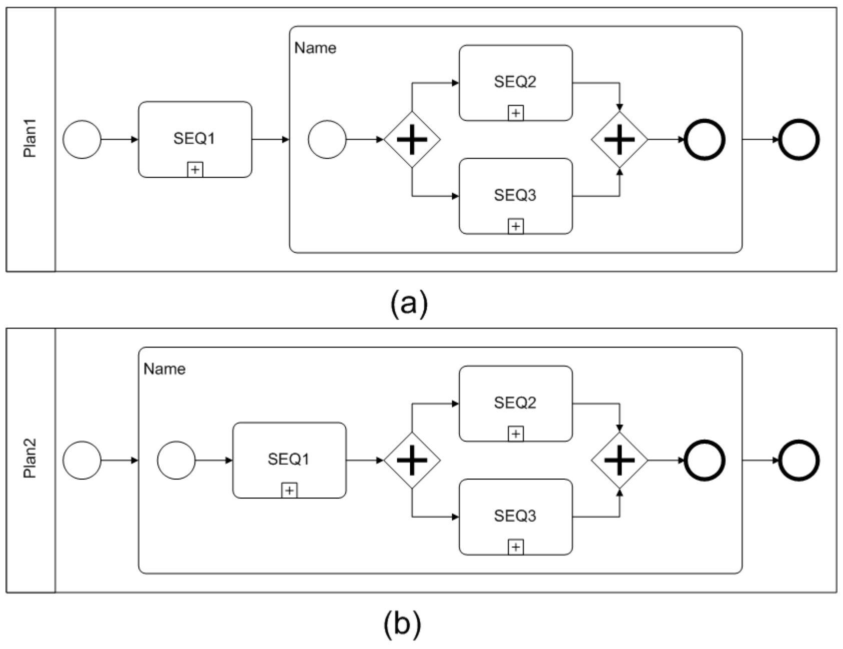


Figure 19: Two BPMN diagrams modelling semantically equivalent observation workflow.

```
[Event (Id "SEQ1") (Pa START), Event (Id "SEQ2") (Pa (Id "SEQ1")),
 Event (Id "SEQ3") (Pa (Id "SEQ1")),
 Event NORMAL_STOP (All [Pa (Id "SEQ2"),Pa (Id "SEQ3")])]
```

Although applying the function `w2b` over this *OWorkflow* definition will yield the diagram in Figure 19(a), one would like to know if applying the function `b2w` over the two diagrams will yield the same *OWorkflow* definition. The formal semantics of BPMN in CSP [16] allows us to show that these two diagrams are in fact semantically equivalent, by model checking the following failures refinement assertions:

$$PLAN1 \sqsubseteq_F PLAN2 \wedge PLAN2 \sqsubseteq_F PLAN1$$

where $PLAN1$ and $PLAN2$ are CSP processes describing the semantics of the BPMN diagrams in Figure 19(a) and Figure 19(b) respectively. This simple example leads to the following definition on *scoping* described in Section 4.5.

Definition 2 *Given a BPMN diagram B describing some observation workflow, B is* ***properly scoped*** *if for all the scoping subprocess states SC defined in B, the outgoing transition of the start state of each scoping state in SC matches an incoming transition of a decision gateway state,* agate *or* xgate, *and the incoming transition of the end state of each scoping state in sc matches an outgoing transition of a decision gateway state.*

6 Conclusion

Specifications of long running empirical studies are complex; the production of a complete specification can be time consuming and prone to error. We have described a graphical method to assist this type of specification. We have introduced an observation workflow model *OWorkflow* suitable for specifying empirical studies, which then can be populated onto a calendar for scheduling, and described bidirectional transformations, which allow empirical studies to be constructed graphically using BPMN, and to be simulated and partially automated as BPEL processes. The transformation also induces a behavioural semantics upon *OWorkflow*, and we have described the use of the semantics to formalise scoping in the transformation process.

To the best of our knowledge, this paper describes the first attempt to apply graphical workflow technology to empirical studies and calendar scheduling, while large amounts of research have focused on the application of workflow notations and implementations to "in silico" scientific experiments. Notable is Ludäscher et al.'s Kepler System [7], in which such experiments are specified as a workflow graphically and fully automated by interpreting the workflow descriptions on a runtime engine. On the other hand we employ BPMN as a graphical notation to specify and graphically visualise experiments and studies that are typically long-running and in which automated tasks are often interleaved with manual ones. Studies such as clinical trial would also include "in vivo" intervention. Furthermore, our approach targets studies that are usually recorded in a calendar schedule to assist administrators and managers. Similarly, research effort has been directed towards effective planning of *specific types* of long running empirical studies, namely clinical trials and guidelines. Notable is Modgil and Hammond's Design-a-Trial (DaT) [8]. DaT is a decision support tool for critiquing the data supplied specifically for randomized controlled clinical trial specification based on expert knowledge, and subsequently outputting a protocol describing the trial. DaT includes a graphical trial planner, which allows description of complex procedural contents of the trial. To ease to complexity of protocol constructions, DaT uses macros, common plan (control flow) constructs, to assist trial designers to construct trial specification.

Future work will include extending our observation workflow model for more detail specifications of work units, such as temporal and procedural information, thereby allowing study plans to be verified against specifications of the relationship between work units and observations.

References

[1] Business Process Execution Language for Web Services, Version 1.1., May 2003. `http://www.ibm.com/developerworks/library/ws-bpel`.

[2] R. Calinescu, S. Harris, J. Gibbons, J. Davies, I. Toujilov, and S. Nagl. Model-Driven Architecture for Cancer Research. In *Software Engineering and Formal Methods*, Sept. 2007.

[3] Clinical Trials Management Tools. University of Pittsburgh, `http://www.dbmi.pitt.edu/services/ctma.html`.

[4] P. Hammond, M. J. Sergot, and J. C. Wyatt. Formalisation of Safety Reasoning in Protocols and Hazard Regulations. In *19th Annual Symposium on Computer Applications in Medical Care*, Oct. 1995.

[5] S. Harris and R. Calinescu. CancerGrid clinical trials model 1.0. Technical Report MRC/1.4.1.1, CancerGrid, 2006. `www.cancergrid.org/public/documents`.

[6] B. Kiepuszewski. *Expressiveness and Suitability of Languages for Control Flow Modelling in Workflows*. PhD thesis, Queensland University of Technology, Brisbane, Australia, 2002.

[7] B. Ludascher, I. Altintas, C. Berkley, D. Higgins, E. Jaeger-Frank, M. Jones, E. Lee, J. Tao, and Y. Zhao. Scientific Workflow Management and the Kepler System. *Concurrency and Computation: Practice & Experience, Special Issue on Scientific Workflows*, 2005. to appear.

[8] S. Modgil and P. Hammond. Decision support tools for clinical trial design. *Artificial Intelligence in Medicine*, 27, 2003.

[9] Object Management Group. *BPMN Specification*, Feb. 2006. `www.bpmn.org`.

[10] Object management group. `www.omg.org`.

[11] OMG. *Business Process Modeling Notation (BPMN) Specification*, Feb. 2006. `www.bpmn.org`.

[12] C. Ouyang, W. M. P. van der Aalst, M. Dumas, and A. H. M. ter Hofstede. Translating BPMN to BPEL. Technical Report BPM-06-02, BPM Center, 2006.

[13] J. Recker and J. Mendling. On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages. In *Proceedings 18th International Conference on Advanced Information Systems Engineering*, pages 521–532, 2006.

[14] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[15] S. White. Using BPMN to Model a BPEL Process. BPTrends, 2005. Available at `www.bptrends.com`.

[16] P. Y. H. Wong and J. Gibbons. A Relative-Timed Semantics for BPMN, 2008. Submitted for publication. Extended version available at `http://web.comlab.ox.ac.uk/oucl/work/peter.wong/pub/bpmntime.pdf`.

[17] XML Schema Part 2: Datatypes Second Edition, Oct. 2004. `http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/`.