International Journal of Advances in Software Engineering & Research Methodology– IJSERM
Volume 2 : Issue 1 [ISSN : 2374-1619]
Publication Date: 30 April, 2015

An application of a four-way framework for validating a specification:

Animating an Object-Z specification using Prolog

Cyrille Dongmo, John Andrew Van der Poll.

***Abstract* — A great deal of the benefits of formal methods stems from their ability to rigorously and precisely specify, at an initial stage, the requirements of the system being developed. Errors in requirements are detected and eliminated earlier and important properties of the system can be formally established thereby, allowing the analysis of the behaviour of the system before the design. These benefits come at two significant prices: firstly, due to its rigour and the level of details, the specification process is a difficult and costly exercise. Secondarily, a formal specification becomes exploitable when it is carefully validated. The search for appropriate validation guidelines, frameworks, methods and techniques is a continuous endeavour of researchers especially with techniques such as Object-Z for which tool support are still very scarce. This paper follows a 4-way framework for validating a specification, to validate an Object-Z specification. During the validation, a mechanism is proposed to translate the specification into Prolog facilitating its animation. A case study is used to illustrate the approach.**

***Keywords*—Formal Specification, Specification validation, 4-way framework, Object-Z, Animation, Prolog.**

I. Introduction

Mathematical approaches to software development are becoming increasingly popular in both academia and industry. Formal requirements specification and the specification validation are two important and challenging phases. The precision, rigour and the level of detailed analysis expected from a formal specification makes the process of transforming informal descriptions of the initial user requirements into mathematical-like expressions a tedious and difficult task. Similarly, the validation of a formal specification, aimed at establishing the correctness of the proprieties of the system being specified, is equally demanding.

Some of the most rigorous and costly validation approaches, e.g. automated proofs are by means of theorem provers. Such approaches involve the mathematical formulation of desirable properties of the system as theorems of which the correctness are demonstrated by means of specialised software, e.g. theorem provers [9]. Animation is another technique in validating a specification and despite criticisms raised against specification animation for not being rigorous enough, research in favour of animating formal specifications has been abundant. Amongst the most prominent reasons put forward in favour of animation is the ability to make the complex nature of mathematical notations transparent, thereby facilitating discussions between developers, users and other stakeholders [10] [15].

This paper is an extension of research, presently conducted, in which we suggested a means to exploit enterprise organograms to address the challenge of scope delimitation in goal and requirements analysis. The model proposed in our previous work, as well as the algorithms to manipulate the model, presented next as a case study, is formalised as an Object-Z specification and subsequently validated. We illustrate how existing Z animations with Prolog can usefully be adapted to animate Object-Z specifications.

II. Case study

Consider the organogram of a college in Figure 1 to which business objectives and some relationships between such objectives are defined to facilitate IT goal/requirements elicitation.

Each node of the organogram (which may be viewed as a directed graph) is either a decisional element (e.g. a director's office) with operational elements attached to it, or simply an operational element (a leaf). An IT project initiated within the college aims to produce a tool to support activities either at a decisional or at an operational level, hence contributing to achieve the college's business objectives. In general, objectives of components at a lower level in the hierarchy of the organogram are sub-objectives of the objectives of the components at a higher level. For a given set of objectives to be supported by an IT project, two search strategies are defined to traverse the organogram to systematically identify all the components (decisional or operational) within the college that may need to be investigated during the requirements elicitation phase. These are horizontal (cf. breadth-first) and vertical (cf. depth-first) searches.

The **horizontal search** purposes to identify, on the basis of horizontal relationships between objectives, nodes within the same domain or sub-domain, which objectives directly or

Cyrille Dongmo
School of computing/ College of Science Engineering and Technology / University of South Africa (Unisa)
South Africa

John Andrew Van der Poll
Graduate School of Business Leadership (SBL) / University of South Africa (Unisa)
South Africa

indirectly contribute to the achievement of other objectives of the system being analysed.

A **vertical search** on the other hand, traverses the organogram, one level up or down of the hierarchy of the organogram to identify nodes with at least one objective either obtained by the refinement of some of the objectives of the system being analysed, or from which some of the objectives of the system were derived by refinement. For example, Research objectives at the college level are achieved through the research activities at the dean office, as well as the achievement of research objectives at School_1, School_2 and School_3.
Assume we are requested to produce an Object-Z specification to address the following stakeholders' requirements:

1) Model the organogram of the case study,
2) Specify a mechanism for the horizontal search,
3) Specify a mechanism for the vertical search,
4) Facilitate the feasibility analysis of the project.

A. *College organogram*

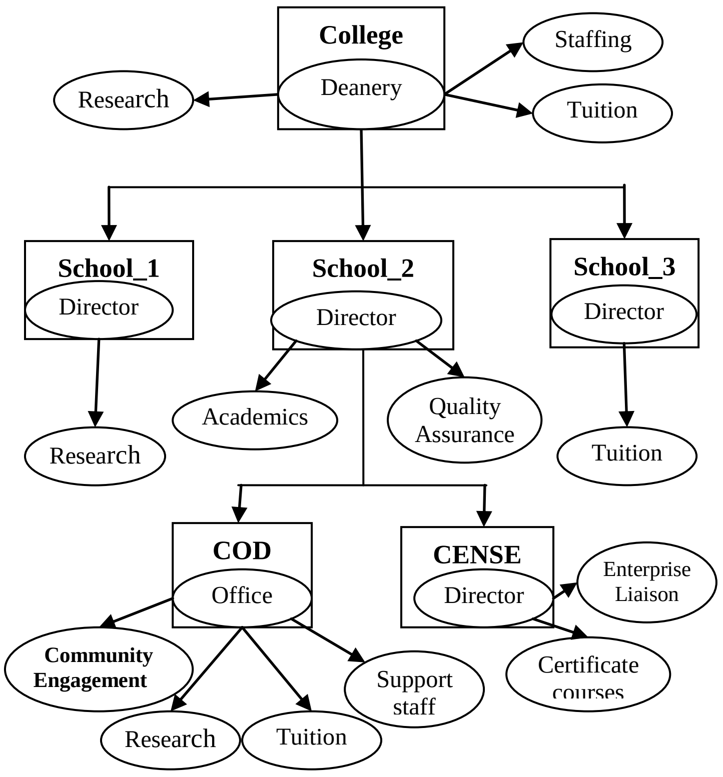

Figure 1: College organogram

To proceed with the Object-Z specification of the college organogram and the algorithms to manipulate it, we next present the graph model of the organogram followed by the algorithms. The first step is to label the nodes as in Table 1.

Table 1 - Labelling the nodes of the organogram

Node label (identifier)	Description of each node		
	Domain	*Management*	*Operational element*
CD	College	Deanery	Dean's office
CR	Colllege	Deanery	Research
CS	College	Deanery	Staffing
CT	College	Deanery	Tuition
S1	College	School_1	Director's office
S2	College	school_2	Director's office
S3	College	school_3	Director's office
S1R	School_1	Director's office	Research
S2A	School_2	Director's office	Academics
S2Q	School_2	Director	Quality assurance
S2D	School_2	COD	Office
S2C	School_2	CENSE	CENSE office
DCE	School_2	COD	Community Engagement
DR	School_2	COD	Research
DT	School_2	COD	Tuition
DS	School_2	COD	Support staff
CEC	School_2	CENSE	Certificate courses
CEE	School_2	CENSE	Enterprise liaison
S3T	School_3	Director	Tuition

A graph model of the organogram is therefore given by the sets of nodes V_{fig1} and the set of edges E_{fig1}.

$$V_{fig1} = \left\{ \begin{array}{l} CD, CR, CS, CT, S1, S2, S3, S1R, S2A, S2Q, \\ S2D, S2C, DCE, DR, DT, DS, CEC, CEE, S3T \end{array} \right\}$$

and

$$E_{fig1} = \left\{ \begin{array}{l} CD \mapsto CR, CD \mapsto CS, CD \mapsto CT, CD \mapsto S1 \\ CD \mapsto S2, CD \mapsto S3, S1 \mapsto S1R, S2 \mapsto S2A \\ S2 \mapsto S2Q, S2 \mapsto S2D, S2 \mapsto S2C, S2D \mapsto DCE, \\ S2D \mapsto DR, S2D \mapsto DT, S2D \mapsto DS, S2C \mapsto CEC, \\ S2C \mapsto CEE, S3 \mapsto S3T \end{array} \right\}$$

The above simplified model of the organogram serves as illustration of what we want to specify formally.

Some traversal algorithms are defined next.

B. *Organogram Traversal Algorithm (OrTA)*

Algorithm 1.0 – General organogram traversal

```
Input    E, CurV, CurObj, CurHrel, CurVrel
Output   CurV', CurObj', CurHrel', CurVrel',
Initialize (CurV, CurObj, CurHrel, CurVrel)
While (there are vertices v in CurV not coloured black)
    For Each vertex v in CurV
        If color(v) is white then
            'horizontal processing of v
            Apply Algorithm 1.1
            Change v color to grey
        ElseIf Color(v) is grey then
            'vertical processing  of v
            Apply Algorithm 1.2
            Change color of v to black
        End If
    Next vertex
End While
End
```

E denotes the set of directed edges, *CurV* contains the currently identified vertices that may be considered during requirements elicitation. *CurObj* is the set of objectives so far identified and *CurHrel* and *CurVrel* represents, respectively, the horizontal and vertical relationships between the currently identified objectives.

The horizontal analysis: For a given node, the purpose is to identify those objectives of the node that are in a horizontal relationship with the currently identified objectives.

Algorithm 1.1 – Horizontal search

```
Input    V, CurObj, CurHrel
Output   CurObj,CurHrel updated
ListObjectives = ran curObj
For Each Ov in ran ({v}◁ nodeObj)
    For each O in ListObjectives
        If Ov ↦ O ∈ hRel or O ↦ Ov ∈ hRel then
            Add Ov to CurObj
            Add Ov ↦ O or O ↦ Ov to CurHrel
        End If
    Next O
Next Objective
```

The vertical analysis: For the input node, the main purpose is to identify direct predecessors and successors of the node of which the objectives are in a vertical relationship with the objectives of the input node.

Algorithm 1.2 – Vertical search

```
Input    v, CurObj, CurV, E, CurVrel
Output   CurV, CurObj, CurVrel updated
ListObjectives = ran ({v}◁ nodeObj)
[analysing the direct  predecessor of v]
If v has a direct predecessor w (w ↦ v ∈ E) then
    If at least one objective of w is in vertical relationship with
    at one or more objectives of v  then
        Add the vertex w to curV if not yet added
        Add each Ow ↦ Ov of vRel to CurVrel,
            where Ov is an objective of v and Ow
            objective of w.
        Add each Ow to CurObj
    End If
End If
[analysing the direct successors of v]
If v has at least one direct successor  then
    For each direct successor s of v (v ↦ s ∈ E)
        If an objective Os of s is in vertical
        relationship with Ov (Ov ↦ Os ∈ Vrel)
        then
            Add s to CurV is not yet added
            Add Os to CurObj
            Add each Op ↦ Ov of vRel to CurVrel
                where Ov is an objective of v and
                Os objective of s.
        End If
    Next successor
End If
```

Before developing an Object-Z specification of the case study, we give some background on Z, Object-Z and validation.

III. Background

A. *Z and Object-Z*

Z is a formal specification notation based on first-order logic and a strongly-typed fragment of Zermelo–Fraenkel set theory [2] [23]. The main concept to describe static and dynamic behaviours of systems is that of a schema. A generic form of a schema is given next:

```
____SchemaName[list of parameters]____
| [declaration – part]
|---------------------
| [predicate – part]
|_____________________
```

The variables of the schema are defined in the declaration part above the short dividing line while constraints on the variables are included in the predicate part [19]. Operations on standard Z elements and schemas, namely the schema calculus, as well as the construction of state and operation schemas are covered in any good book on Z, e.g. [14] and [21]. Object-Z is an object-oriented extension of Z that encapsulates standard Z schemas in a class structure [6] [22]. A generic form of an Object-Z class schema is shown [4].

```
ClassName [generic parameters]
  [visibility]
  [inherited classes]
  [local definitions]
  [state schema]
  [initial state schema]
  [operations]
```

The visibility list restricts access to specific elements of the class. A list of inherited classes is included as well as local Z definitions. Only one state schema is allowed in a class, which may be initialised. Operations are specified within the class. Concrete examples of Object-Z class schemas are given in subsequent sections. Operations on class schemas used in this work will be explained when they are encountered.

B. *Specification validation*

Validating a formal specification is a tedious task that has been studied by researchers and practitioners over many years. Different levels/categories of validation are employed, each of which addresses a specific aspect of the specification and requires specific tools:

- ***Reviews and Inspection*** – this involves manually checking a specification to detect and correct problems, e.g. Fagan inspections [8][5][7]. This technique is less rigorous and requires a fair amount of human effort.
- ***Parsing and type-checking*** – these two techniques are concerned mainly with detecting errors related to the specification language (syntactic and semantic errors) and aim to ensure the internal consistency of the specification [13] [12]. Most of the tool support available perform both parsing and type-checking [20].
- ***Animation*** – animating a specification involves executing the specification with appropriately selected test data and observing its behaviour [10] [11]. An animation process generally includes two major phases: the transformation of the specification into an executable form, followed by the execution phase. Although animation techniques are less rigorous than formal proofs, they have been widely adopted as a means for prototyping formal specifications.
- ***Mathematical proofs –*** properties of the system are formulated as theorems, of which the proofs are discharged in either an automated or semi-automated fashion by specialized software, namely theorem provers [9] [17].

C. *A 4-way validation framework*

The four-way framework for validating a specification [3], proposes to iteratively validate and amend a specification until the desired quality is obtained. As shown on Figure 2, at each iteration four validation phases are considered; with each phase focused on one important aspect of the specification.

The rightward phase targets the properties of the specification related to the specification language and any associated tool support. The upward validation focuses on the properties of the specifications pertaining to stakeholders' expectations and initial goals. The leftward phase addresses attributes of the application domain, while the downward validation ensures that the specification can eventually lead to the envisioned software product. The order of the phases is not prescriptive, but in this work, we start with the rightward phase to first ensure that the specification is well-formed and can, therefore, serve as input to the available tools.

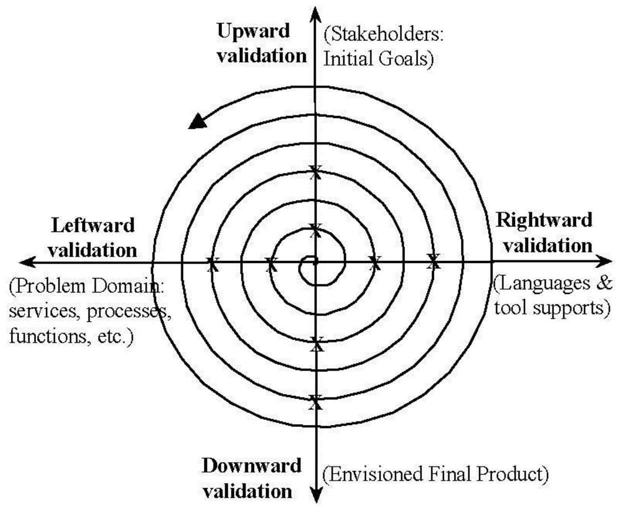
Upward validation
(Stakeholders: Initial Goals)
Leftward validation
(Problem Domain: services, processes, functions, etc.)
Rightward validation
(Languages & tool supports)
Downward validation
(Envisioned Final Product)

Figure 2: Four-way framework for validating a specification [3]

D. *Animating Z/Object-Z specifications*

Although approaches and tools to validate Object-Z specifications, especially animators and theorem provers, are still rather scarce, many more have been developed for Z. Many of the methods for animating Z use Prolog. Two main approaches have been proposed: ***formal program synthesis*** and ***structure simulation*** [25].

Formal program synthesis obtains a Prolog program from the Z specification by means of a direct two-step transformation of Z schemas. First any higher-order Z constructs are rewritten as first-order formulae, and second such first-order formulae are converted into Prolog. The challenge with this approach is that the second step is manual – there is no suitable algorithm to turn the first-order specifications into logic programs [18]. Following *structure simulation*, a Prolog program is created based on the characteristics of the Z specification, which may have been "flattened" by eight (8) guiding rules derived for this approach

[25]. Seven (7) of these rules are adapted in this paper for the Z/Prolog transformation that follows :

(1) For each Z schema, create the following Prolog predicates:

- ➢ *schema_type (L, N)* for state schemas and *schema_op (L, N)* for operations. L is the list of variables associated to schema N.
- ➢ givenset (S, N) where S is a given set and N the given set name.

(2) Possible values of variables in a Z schema are described in the body of the clause using the logical relationship.

(3) Concerning Z types, the given sets characterising the schema are specified first. Each declared (type) variable is represented by two predicates, the one naming, the other giving the type. Decoration of variable names is achieved by Prolog functions; thus s? and s! are named in(s) and out(s), respectively, where s is the base name. Similarly, x and d(x) name a state variable and the post-operational state.

(4) Set operations, such as intersection and union, are (assumed to be) contained in a library of Prolog code. Otherwise, we implement these when needed.

(5) A variable that is existentially quantified in a Z schema's clause appears in the body of the Prolog translation of the clause as a Prolog variable, which does not appear in the declarations of a named variable.

(6) The Prolog translation of the conjunction C of two schemas A and B is obtained as follows: translate the signature of C, which is obtained by merging the signatures of A and B. The Prolog translation of the predicates of A and B are conjoined to obtain that of C. An analogous rule applies to the disjunction of two schemas.

(7) When a schema B is used as a type in the signature of A, during the translation, schema B is conjoined to the signature of schema A.

IV. Guidelines for animating an Object-Z specification in Prolog

A three-fold process is followed for the animation: *Firstly, we unfold the Object-Z classes* to extract the encapsulated Z schemas. Secondly, we *transform each Z element into Prolog* and lastly, *proceed with the execution*.

A. *Unfolding Object-Z classes*

Since an Object-Z specification normally encapsulates standard Z elements with very little modifications, we take advantage of the existing Z transformation guidelines. We unpack each class individually to work on the embedded Z elements. When necessary, an identified Z element may be slightly modified to undo slight changes due to Object-Z transformation.

B. *Transforming Z schemas into Prolog*

We use the above 7 guidelines to transform each Z schema into a Prolog program. As and when necessary, more explanation is provided during the transformation process. The symbol "/" is used to couple related elements within a relation or a function.

C. *Animating the specification*

Two standard questions are used to guide the execution of the specification: Are we building the correct system? (Validation) and are we building it right? (Verification). The first question concerns the validation of the specification against the initial requirements, stakeholder goals (upward validation), as well as constraints from the application domain (leftward validation). The second question addresses the consistency and correctness of the specification (rightward validation. A successful animation of the specification also indicates that the specification can be transformed into operational software (downward validation).

As noted by West [24], an important requirement for a successful animation is the ability to trace back in the specification the source of errors when they occur. This requirement is achieved by creating a Prolog program that mimics the structure of any Z element under consideration; whenever possible, the names used in the specification are conserved in Prolog.

V. Object-Z specification of the case study

Each node of the graph is represented by its identifier. The set of all possible identifiers, as well as business objectives are represented as empty Object-Z classes as shown next [4].

Identifier

ClsOperationalElt
$\upharpoonright(id, name)$

$id : Identifier$
$name : Description$

ClsDecisionalElt
$\upharpoonright(designation, office)$

$designation : Description$
$place : Description$
$office : ClsOperationalElt$

$$\begin{array}{l}
idnode : Node \rightarrowtail\!\!\!\!\!\!+ \; Identifier \\
\hline
\forall\, node : Node \bullet \\
\quad node \in ClsOperationalElt \Rightarrow \\
\qquad idnode(node) = node.id \\
\quad node \in ClsDecisionalElt \Rightarrow \\
\qquad idnode(node) = node.office.id
\end{array}$$

A *node* is either an operational element *or* a decisional element, i.e.:

$Node == ClsOperationalElt \cup ClsDecisionalElt$

$$\begin{array}{l}
\underline{ClsOrganogram} \\
\upharpoonright(edges, nodeobj, HorizSearch, VerticalSearch) \\
[Identifier, Objective] \\
\quad hRel : Objective \leftrightarrow Objective \\
\quad vRel : Objective \leftrightarrow Objective \\
\hline
\quad edges : Identifier \leftrightarrow Identifier \\
\quad nodeObj : Identifier \leftrightarrow Objective \\
\hline
\quad \mathrm{dom}\, nodeObj \subseteq \mathrm{dom}\, edges \cup \mathrm{ran}\, edges \\
\hline
\quad \underline{INIT} \\
\quad edges = \varnothing \land nodeObj = \varnothing \\
\hline
\quad \underline{HorizSearch} \\
\quad id? : Idtentifier, listObj? : Identifier \leftrightarrow Objective \\
\quad listOut! : Identifier \leftrightarrow Objective \\
\hline
\quad \forall\, Ov \in \mathrm{ran}(id? \lhd nodeObj) \bullet \\
\qquad \forall\, O \in \mathrm{ran}\, listObj? \bullet hRel(Ov, O) \Rightarrow \\
\qquad\quad listOut! = listObj? \cup \{Ov \mapsto O\} \\
\hline
\quad \underline{VerticalSearchUp} \\
\quad v? \in nodes \\
\quad curObj?, curObj! \subseteq nodeObj \\
\quad curvRel?, curvRel! \subseteq vRel \\
\quad curhRel?, curhRel! \subseteq hRel \\
\quad CurVetices?, curVertices! :\subseteq nodes \\
\hline
\quad \exists\, p1 \in nodes \bullet p1 \mapsto v? \in edges \Rightarrow \\
\qquad \forall\, P1 \mapsto Op, v? \mapsto Ov \in nodeObj \bullet \\
\qquad\quad Op \mapsto Ov \in vRel \Rightarrow \\
\qquad\qquad curVertices! = curVertices? \oplus \{p1\} \\
\qquad\qquad curObj! = curObj? \oplus \{p \mapsto Op\} \\
\qquad\qquad curvRel! = curvRel? \oplus \{Op \mapsto Ov\} \\
\quad \forall\, p \in nodes \bullet v? \mapsto p \in edges \bullet \\
\quad \forall\, p \mapsto Op, v? \mapsto Ov \in nodeObj \bullet \\
\qquad Ov \mapsto Op \in vRel \Rightarrow \\
\qquad\quad curVertices! = curVertices? \oplus \{p\} \\
\qquad\quad curObj! = curObj? \oplus \{p \mapsto Op\} \\
\qquad\quad curvRel! = curvRel? \oplus \{Ov \mapsto Op\}
\end{array}$$

The class organogram specifies the nodes as a set of identifiers to make it possible to use only this class for the animation and hence avoid extra operations such as *idnode* and concentrate on the two key operations which are: *HorizSearch* and *VerticalSearch*.

VI. Validating the specification

The following techniques are proposed for each of the 4 validations in Figure 2:

Table 2: Planning the validation for one iteration

Iteration			
Phases	***Method***	***Tool***	***Property***
Rightward	Review Parsing and type checking	Manual CZT	Internal consistency
Upward	Animation / Prototype	Prolog	Correctness
Leftward	Domain analysis Animation / Prototype	Prolog	Completeness Applicability
Downward	Arguments Inventory of techniques	manual	Feasibility

The phases and methods in Table 2 are discussed next.

A. *The rightward validation phase*

This phase is carried out to resolve language related errors and ensure that the specification is internally consistent and well-formed. Two approaches are used: informal inspections and automatic type checking.

The inspection of the specification entails: detect contradicting elements within a class, check that each class is well-structured with respect to a manageable size, especially the size of the schema, and check that the formulas in the predicate parts of the state and operation schemas within a class are syntactically correct and intuitively clear to an average reader.

The Object-Z specification in this paper was constructed in a latex document using the OZ.STY macro [1] and type-checked with the Community of Z Tools – CZT version 1.5 [16].

B. *The upward validation phase*

As indicated in The following techniques are proposed for each of the 4 validations in Figure 2:

Table 2, we seek to animate the specification purposing to establish the correctness of the specification regarding the initial goals and/or requirements.

The three (3) empty classes ***Identifier***, ***Objective***, and ***Description***, resulting from *ClsOrganogram* above are each transformed into a given type in Z, which is presented as a clause in Prolog.

The guidelines proposed above are further applied to the class *ClsOrganogram*. The two other classes may be animated in a very similar vein.

a. **Unpacking this class**, we obtain:

(1) Two relations, namely, *hRel*, *vRel*, and a function *idnode* which remain unchanged in Z. Implementing a relation (and a function) in Prolog is rather straightforward.

(2) An abstract state, namely, *stateOrganogram* which is simply the named version of the class state.

(3) Two operation schemas: *HorizSearch* and *VerticalSearch*. Neither of these operations changes the state schema. However, in their Z form; the term ***ΞstateOrganogram*** is included into each of the operation schemas.

Next, is the Prolog implementation of the class organogram.

b. Prolog implementation

1- The horizontal and vertical relationships (hRel & vRel)

Sample data used in this paper are given in Table 3. The first column contains node identifiers and the second column the objectives at each node. These columns are implemented as given sets. The last two columns provide for the vertical and horizontal relationships between the objectives. An element x ↦ y ∈ vRel (y is obtained from x by refinement) is encoded in Prolog as x/y. x ↦ y ∈ hRel means that x supports y or y needs x to be achieved.

2- The state schema *stateOrganogram*

The variables in the signature of the state schema are: *edges*, and *nodeObj*. The clause varname(_ , _) is used in Prolog to associate data to these variables. The predicate ***schema_type ([ ], schema_name)*** is used to implement any state schema. The first argument contains the signature and the second the name of the state space. Since the operations need access to the signature of the state schema, the Prolog assert operation is used to save them during the execution of schema_type.

```
varname(_, edges).
varname(_, nodeob).

schema_type([Edges, Obj], stateorganogram):-
        %given sets
        givenset(Ns, identifier),
        givenset(Gs, objective),

        %variables names
        varname(Edges, edges),
        varname(Obj, nodeob),

        % variables' definition
        validedges(Edges),
        validnodeobj(Obj),

        %Predicate
        findall(X, element(X/_, Edges), DE),
        rem_dups(DE, DomE),
        findall(D, element(_/D, Edges), RE),
        rem_dups(RE, RanE),
        union(DomE, RanE, UNodes),

        findall(X,element(X/_,Obj), DomOb),
        rem_dups(DomOb,Dob),
        subset(Dob,UNodes).
```

The set operations: rel(L,N) (test a relation), element(x, L), subset(X, L), rem_dups (X, NewX) (remove duplicates), etc. were included in the program.

3- The operations: *horizSearch* and *verticalSearch*

Each Z operation schema is implemented as:
Schema_op (L, N)

c. Executing the Prolog program

Table 3 - Sample data

Node	objectives	Vertical relationship(vRel)	Horizontal relationship(hRel)
Cd	cd_o1, cd_o2, cd_o3		
Cr	cr_o1, cr_o2, cr_o3		cr↦cd_o1, cr_o2↦cd_o2
Cs	cs_o1, cs_o2		cs_o1↦ cd_o2, cs_o2↦ cd_o2
Ct	ct_o1		ct_o1 ↦cd_o1
s1	s1_o1, s1_o2	cd_o1↦s1_o1, cd_o2↦s1_o2	
s2	s2_o1, s2_o2	cd_o1↦s2_o1, cd_o2↦ s2_o2	
s3	s3_o1, s3_o2	cd_o1↦s3_o1, cd_o2↦s3_o2	
s1r	s1r_o1, s1r_o2		s1_o1↦ s1r_o1
s2a	s2a_o1,s2a_o2, s2a_o3	s2_o1↦s2a_o1, s2_o2↦ s2a_o2	
s2q	s2q_o1,s2q_o2, s2q_o3	s2_o2↦ s2q_o1	s2q_o2 ↦ s2a_o1, s2q_o3 ↦ s2d_o2, s2q_o3 ↦ s2c_o2
s2d	s2d_o1, s2d_o2, s2q_o3, s2q_o4	s2_o1 ↦ s2d_o1, s2_o2 ↦ s2d_o2	
s2c	s2c_o1, s2c_o2	s2_o2 ↦ s2c_o1	
Dce	dce_o1	s2d ↦dce_o1,	
Dr	dr_o1, dr_o2, dr_o3	S2d_o2 ↦ dr_o2, s2d_o2 ↦dr_o3	
Dt	dt_o1, dt_o2	S2d_o1 ↦dt_o1	
Ds	ds_o1, ds_o2, ds_o3		ds_o1 ↦ dr_o2, ds_o2 ↦ dr_o3, ds_o3 ↦ dt_o1, ds_o1 ↦ dt_o2
Cec	cec_o1, cec_o2, cec_o3	s2c_o1 ↦ cec_o1, s2c_o2 ↦ cec_o3	
Cee	cee_o1, cee_o2		Cee_o1↦cec_o2, cee_o2 ↦cec_03
s3t	s3t_o1, s3t_o2	cd_o1 ↦ s3t_o1	

1- Testing the initial state

Initially, both the two components of the state schema denote empty sets.

```
??- schema_type([[], [] ], stateorganogram).
```

International Journal of Advances in Software Engineering & Research Methodology– IJSERM
Volume 2 : Issue 1 [ISSN : 2374-1619]
Publication Date: 30 April, 2015

```
yes
??-
```

2- Verifying the state schema

We test the state schema with the data for the sub-graph rooted at School_2 (node id = s2).

```
??- schema_type([[s2/s2a, s2/s2d, s2/s2c, s2d/dce, s2d/dr,
      s2d/dt, s2d/ds, s2c/cec, s2c/cee], [s2/s2o1, s2/s2o2,
      s2a/s2ao1, s2q/s2qo1, s2d/s2do1, s2d/s2do2,
      s2c/s2co1, dce/dceo1,  dr/dro1, dt/dto1, dt/dto2,
      ds/dso1, cec/ceco1, cee/ceeo1]], stateorganogram).

      yes

??-
```

Each sub-graph, as well as the whole organogram was tested in a similar fashion.

3- Testing the horizontal search operation

```
% Predicate to select a tuple for output
hsel(Ov, Oi, V, List):-
        nodeobj(V/Ov),
        element(Oi, List),
        ( hrel(Ov/Oi) ;
        hrel(Oi/Ov) ).

varname(_,  in(id)).
varname(_, inout(curobj)).
varname(_, inout(curHrel)).

schema_op([Nodeid, CurObj, CurHrel],horizsearch):-
        %Given sets
        givenset(Ns, identifier),
        givenset(Gs, objective),

        %variables' names
        varname(Nodeid, in(id)),
        varname(CurObj, inout(curobj)),
        varname(CurHrel, inout(curobj)),

        %display inputs
        write('Input-Current list of objectives: '),
         write(CurObj), nl,
        write('Input-Current list of horizontal relationships: '),
         write(CurHrel), nl,

        %find the horizontal relationships for the input node
        findall(Ov/Oi,hsel(Ov,Oi,Nodeid,CurObj),Od),
        union(Od,CurHrel,CurHrelo),

        %Add input node' objectives participating to horizontal relation
        findall(X,element(X/_,Od),Obs),
        union(Obs,CurObj,CurObjo),

        %display outputs
        write('Output-Current list of objectives: '), write(CurObjo),nl,
        write('Output-Current list of horizontal relationships: '),
         write(CurHrelo).
```

The predicate *hsel(Ov, Oi, Nodeid, CurObj)* determines the sub-set of the horizontal relationships that relate the objectives *Ov* of the selected node, which identifier is in Nodeid, to the objectives currently identified and kept in *CurObj*.

```
Amzi! Prolog  Listener
Amzi Prolog Listener 5.0.18h Windows
Aug 21 2000 20:19:21
Copyright (c) 1987-2000 Amzi! inc.

?-reconsult('C:\\dongmc\\User Data\\Studies\\PHD Package\\Journal And
Articles\\ACEC 2014\\Prolog\\operationalelt.pro')
yes
?- schema_op([s2q, [s2ao1, s2do2], [] ], horizsearch).
Input-Current list of objectives: [s2ao1, s2do2]
Input-Current list of horizontal relationships: []
Output-Current list of objectives: [s2qo2, s2qo3, s2ao1, s2do2]
Output-Current list of horizontal relationships: [s2qo2 / s2ao1, s2qo3 / s2do2]
      yes
?-
```

The node s2q (school_2 quality assurance) is used as input. Before the execution of the operation, the current list of horizontal relationships is empty. After the execution, the system has successfully determined two relationships: *s2qo2 / s2ao1* and *s2qo3 /s2do2*. The first relationship indicates, for e.g., that the objective *s2qo2* of the quality assurance is meant to support or reinforce the objective *s2do2* that needs to be achieved by the academics in the school.

We have so far presented the implementation and testing of the operation HorizSearch. The vertical search operation, *VerticalSearch*, can be implemented in the same vein. Since these two operations are the core dynamic components of the system being specified, a successful animation/execution of each of them constitutes a major step towards validating the proposed model and consequently, the correctness of the specification as planned in The following techniques are proposed for each of the 4 validations in Figure 2:

Table 2. However, a complete validation would include the loop invoking them (algorithm 1.0), as well as operations to update the different components of the input graph, e.g. adding/removing nodes to/from the graph.

C. *The leftward validation phase*

Having demonstrated the correctness of the system with regard to stakeholders' expectations, that is, for the system to produce appropriate outputs, we need to ensure that the specification also takes into consideration some domain related constraints that are not always explicitly included in the initial requirements. For example, it would be relatively easy to show that our specification will work well for a system with only one PC, or in a network environment with a few computers. But, if it is to operate in a large (International) network environment, or in a distributed system environment, or is to serve in the cloud, the current specification has to be updated to include for example, language issues, constraints related to distributed system (communication), etc.

The animation at this stage would consist of generating appropriate test data to estimate for example the response time, the ability of the system to integrate different languages, etc.

D. The downward validation phase

Having demonstrated the correctness of the specification relative to stakeholders' expectations and adequacy with the application domain, this phase aims to establish the operational ability/feasibility of the specification. Since, at this stage, the only resource at hand is the specification, an approach to demonstrate that an operational system, with the required properties, can be effectively constructed, may proceed by: identifying an appropriate design/refinement process, as well as the methods/techniques and technology needed.

Based on the success of the above animation, it may be relatively easy to derive a desktop application from the specification. However, it would not be the case if the envisioned system is a web application or a cloud service.

VII. Conclusion and Future work

This paper proposed an approach to formally specify the guidelines to construct enterprise organograms in a bottom-up fashion and the transformation into useful models that can be exploited in goal and requirements elicitation phases to identify vital sources of information within the entire organisation. Our approach also proposes strategies to manipulate the model and derive the necessary information in a simplistic manner. An Object-Z specification of the approach was presented to use various benefits of formal methods in the model. The specification was validated using an enhanced spiral model to ensure its correctness, or to identify any aspects still in need of attention. The specification facilitates the implementation of our method and further manipulation thereof. The implementation in this paper was in Prolog, but it could equally have been in some high level Object-Oriented programming language, e.g. Java or a .Net language.

The main advantage of the proposed method stems from the simplicity and availability of enterprise organograms to which appropriate information may be cautiously added to construct flexible and lightweight enterprise models vital to goal and requirements elicitation.

Being an acyclic graph, the vast theory of tree searching algorithms, heuristics and associated complexities may usefully be applied to organogram construction and maintenance. Suitable and economically feasible techniques for the horizontal and vertical traversals of these structures could be investigated for implementation. Knowledge representation and management techniques ought to be evaluated to determine their applicability to carry the goals and requirements in the hierarchy of the organogram.

Based on the proposed guidelines, the Object-Z specification and the above search strategies call for an expert system tool to traverse the organogram and elicit the objectives and services within the sub-domain, as well as the resources allocated. With appropriate interfaces, such a tool could additionally assist an enterprise in maintaining its objectives, services and resources.

References

[1] Edward B. Allen. Typesetting Technical Reports that Include Z Specifications Using LaTeX, 2006.

[2] Jonathan Bowen. *Formal Specification and Documentation Using Z: A Case Study Approach*. International Thomson Computer Press, London / Boston, revised 2003. ISBN 1-85032-230-9.

[3] Cyrille Dongmo and John A. van der Poll. A Four-Way Framework for Validating a Specification. In Paula Kotze, Aurona Gerber, Alta van der Mervwe, and Nicola Bidwell, editors, *SAICSIT*, pages 46–59. ACM PRESS, 2010. ISBN 978-1-60558-950-3.

[4] Cyrille Dongmo and John Andrew van der Poll. Addressing the construction of Z and object-z with use case maps (ucms). *International Journal of Software Engineering and Knowledge Engineering*, 24 (2): 285, 2014. doi: 10.1142/S0218194014500120. URL http://dx.doi.org/-10.1142/S0218194014500120.

[5] E. P. Doolan. Experience with fagan's inspection method. *Software: Practice and Experience*, 22 (2): 173–182, 1992. ISSN 1097-024X. doi: 10.1002/spe.4380220205. URL http://dx.doi.org/10.1002/-spe.4380220205.

[6] Roger Duke and Gordon Rose. *Formal Object-Oriented Specification Using Object-Z*. Macmillan, Basingstoke, 2000. ISBN 0333801237.

[7] Kamsties Erik, Berry Daniel M., Paech Barbara, E. Kamsties, D. M. Berry, and B. Paech. Detecting Ambiguities in Requirements Documents Using Inspections. In *Proceedings of the First Workshop on Inspection in Software Engineering (WISE&apos;01)*, pages 68–80, 2001.

[8] Michael E. Fagan. Design and Code Inspections to Reduce Errors in Program Development. *IBM Systems Journal*, 15 (3): 182 – 211, 1976.

[9] Leonardo Freitas. Proving theorems with z/eves. *Appendix A*, 1 (1), 2004.

[10] Jeff Gray and Stephen Schach. Constraint animation using an object-oriented declarative language. In *Proceedings of the 38th Annual on Southeast Regional Conference*, ACM-SE 38, pages 1–10, New York, NY, USA, 2000. ACM. ISBN 1-58113-250-6. doi: 10.1145/1127716.1127718. URL http://doi.acm.org/10.1145/-1127716.1127718.

[11] W. Hasselbring. Animation of object-z specifications with a set-oriented prototyping language. In J.P. Bowen and J.A. Hall, editors, *Z User Workshop, Cambridge 1994*, Workshops in Computing, pages 337–356. Springer London, 1994. ISBN 978-3-540-19884-0. doi: 10.1007/978-1-4471-3452-7_20. URL http://dx.doi.org/10.1007/978-1-4471-3452-7_20.

[12] Xiaoping Jia. Ztc: A type checker for z notation–user's guide. *DePaul University, Institute for Software Engineering, Department of Computer Science and Information Systems, Chicago, Illinois, USA, version*, 2, 1998.

[13] Wendy Johnston. A Type Checker for Object-Z. Technical report, Department of Computer Science, The University of Queensland Australia, 1996.

[14] David Lightfoot. *Formal Specification Using Z*. Grassroots Series. Palgrave, 2nd edition, 2001.

[15] Shaoying Liu and Hao Wang. An automated approach to specification animation for validation. *Journal of System Software*, 80 (8): 1271–1285, August 2007. ISSN 0164-1212. doi: 10.1016/j.jss.2006.12.540. URL http://dx.doi.org/10.1016/j.jss.2006.12.540.

[16] Petra Malik and Mark Utting. CZT: A Framework for Z Tools. In *International Conference of Z and B Users (ZB 2005)*, pages 65–84. Springer, 2005.

[17] W. McCune. Prover9 and Mace4. http://www.cs.unm.edu/~mccune/prover9/, 2005–2010.

[18] Katsuhiko Nakamura. Introduction to logic programming • by christopher j. hogger. *New Generation Computing*, 3 (4): 487–487, 1985. ISSN 0288-3635. doi: 10.1007/BF03037083. URL http://dx.doi.org/-10.1007/BF03037083.

[19] Gerard O'Regan. *Mathematical Approaches to Software Quality*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006. ISBN 184628242X.

[20] Tim Parker. *TOZE-A Graphical Editor for the Object-Z Specification Language with Syntax and Type Checking Capabilities*. PhD thesis, University of Wisconsin-La Crosse, 2008.

[21] Ben Potter, David Till, and Jane Sinclair. *An Introduction to Formal Specification and Z*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 1996. ISBN 0132422077.

[22] Graeme Smith. *The Object-Z specification language*. Kluwer Academic, Boston, 2000. ISBN 0792386841.

[23] J Michael Spivey. The z notation: a reference manual. international series in computer science, 1992.

[24] Margaret Mary West. *Issues in Validation and Executability of Formal Specifications in the Z notation*. PhD thesis, School of Computing, 2002.

[25] M.M. West and B.M. Eaglestone. Software development: two approaches to animation of z specifications using prolog. *Software Engineering Journal*, 7 (4): 264–276, 1992. ISSN 0268-6961.