Model Evolution: Comparative Study between Clafer and Textual UML

Dina O. Zayan
Generative Software
Development Lab
University of Waterloo
Canada
dzayan@gsd.uwaterloo.ca

ABSTRACT
This paper presents a comparative study between two modeling languages; Clafer and Textual UML. In this work, we present our motivation for domain modeling, we perform a pilot study to provide preliminarily results about the differences between both modeling languages with respect to structural model evolution at the requirements stage. We conclude with lessons learnt and considerations to take into account with the experimental design for the empirical study.

Categories and Subject Descriptors
D.3 [**Software**]: Modeling Languages.

General Terms
Structural Modeling, Languages, Constraints, Evolution.

Keywords
USE, OCL, UML, Clafer, Evaluation.

1. INTRODUCTION
The efficient and effective employment of model-based software engineering starts with creating and evolving models that capture elements from the problem domain precisely. In this paper, we are interested in domain modeling. A domain model can be thought of as a representation of the concepts from a domain of interest. It can be used to effectively validate the understanding of the problem domain among various stakeholders. A high-fidelity domain model could also serve as a domain metamodel which in turn can be used to create a Domain Specific Language (DSL).

Domain models always evolve during the requirements stage since the domain understanding changes frequently during elicitation and with increased understanding of the problem. In this paper, we consider two languages that can be used for creating highly detailed and accurate domain models at the requirements stage: Clafer [3] and UML [5, 6].

Clafer (**Cla**ss, **fe**ature, **r**eference) is a general purpose lightweight structural modeling language developed at the Generative Software Development Lab, University of Waterloo. Clafer is being designed to support domain modeling, requirements elicitation and specification. We choose UML to be the second modeling language in our comparison since it is currently the predominant modeling language in the software industry. However, since Clafer has a textual notation, we select a textual variant of UML: USE (UML-based Specification).

The rest of the paper is organized as follows: section 2 introduces the model structure of Clafer and textual UML, section 3 describes the employed methodology for this research, section 4 provides the analysis of the developed models, section 5 provides the evaluation of the preliminarily results out of this pilot study as well as possible threats to their validity, and finally section 6 provides the conclusions. Complete Clafer and UML models are included in appendices at the end of the paper

2. CLAFER AND TEXTUAL UML

2.1 Textual UML Model
A specification written in USE is a textual description based on UML class diagrams and Object Constraint Language (OCL) constraints. To define a USE specification, you need a text editor. In USE, every UML model has a model name and an optional body [5, 6]. The model body is divided into four sections; Enumeration Definitions, Class Definitions, Association Definitions, and Constraints.

All enumerations have to be placed on top of the model directly below the model name. Class declarations form the next section of the model. Class declarations contain definitions of attributes. The third section includes the associations. There are three types of associations that could be included in the model; regular association, composition, and aggregation. USE supports the definition of association names, role names and association end multiplicities. The fourth and final section of the model represents the OCL constraint definitions. Comments could be added to any section in the model, but must be preceded with --.

2.2 Clafer Model
Clafer is a class modeling language with first class support for feature modeling [3]. It provides uniform syntax and semantics to class and feature models. A Clafer model consists of clafer[1] definitions and constraints. The model is not divided into separate sections like in USE. Clafers express the domain concepts in the model and the possible variability among them with the help of nested constraints. There are two types of clafers:

1. *Abstract Clafers:* An abstract clafer declaration defines a new type. The set of abstract clafers represent the metamodel of the domain or the system under test.
2. *Concrete Clafers:* A concrete clafer represents a possible set of instances/configurations of an abstract one.

[1] The name "Clafer" refers to the modeling language. The expressions "clafer", "clafers", "children/parent clafer" refer to the elements of the model written in Clafer language.

Clafer expresses the composition relationship by indenting child clafers under a parent one. There are several features associated with a Clafer model such as:

1. *Hierarchy/Nesting:* Clafer supports hierarchies by means of indentation.
2. *Clafer Cardinality:* defines how many instances of a given clafer can appear as children of the parent clafer.
3. *Group Cardinality:* defines how many children of a given clafer can be instantiated.
4. *Inheritance:* Clafer supports single inheritance. Both abstract and concrete clafers inherit everything from their superclafers.
5. *References:* Clafer supports references as a means of defining relations between clafers. Clafer does not directly support associations.

3. METHODOLOGY

In this section we present the methodology applied to the design of the pilot study in a way that would benefit a longer term empirical study, and to make a preliminary evaluation of Clafer and Textual UML as structural modeling languages with respect to their support for model evolution [4]. We relied on the principles from the Goal Question Metric (GQM) methodology [1]. The methodology deals with three levels:

1. *Conceptual Level;* Specification of goals. Our goal was to design an empirical study to compare between Clafer and Textual UML in terms of their support for structural model evolution at the requirements stage.
2. *Operational Level;* Formalization of goals assessment in terms of research questions.
   - 2.1 Which language better supports structural modeling and expressing constraints?
   - 2.2 Which language better supports model evolution?
   - 2.3 What are the strengths and weaknesses of each language in general and with regard to model evolution in specific?
   - 2.4 What kind of tool support would help with model evolution?
   - 2.5 What are the recommendations for the design of Clafer?
3. *Quantification Level;* Specification of a set of evaluation criteria to address the defined research questions –presented in Section 3.1.

In order to design an experiment, we needed to have a hypothesis or initial claims about each modeling language that we wanted to verify, thus we decided to perform this pilot study to get preliminary results in the form of answers to the first three research questions. Moving forward with the pilot study, first a choice had to be made about which textual UML modeling specification we were going to use in the study. Our research explored different textual representations for UML; the Kernel MetaMetaModel (KM3) specification, Human Usable Textual Notation (HUTN), EMFText, and the UML-Based Specification (USE).

We modeled a Book example using Clafer, and each of the UML textual representation candidates to determine the features of each, differences among them, and to decide which specification is better suited to represent UML class diagrams in a textual notation. We were looking for common class diagram features support including; class declarations, associations, attribute/reference cardinality, integration of OCL constraints, inheritance, object diagram support. HUTN only supports textual representation for class instances. EMFText does not support associations. KM3 does not support the integration of OCL constraints into the same model neither the modeling of associations, thus we chose the USE specification. However, we added a few extensions to USE to conform to standard UML class diagram modeling: we allowed cardinalities and non-primitive types for attributes (i.e., references).

Next, we selected the type and scope of the system to be used in the pilot study. We wanted a real world example that has a relatively big scope to serve as a reference for evaluating Clafer even after completing the pilot study. Using an independent example helped us to avoid any bias if we had designed our own example. A real world example would not only have a significant amount of requirements that would greatly help model evolution, but it would also help us perform and experience the process for which Clafer is intended: taking a general description of a system, and constructing a formal version of it using Clafer. We used Oracle Retail Merchandise Financial Planning (Oracle MFP) System documents as the reference from which we elicited the requirements and formalized them into a requirements document representing the system under test [7].

After finishing the models completely, we checked for syntax/logical errors by their supported tools. We successfully parsed the Clafer models using Clafer tools, but the Instance Generator (ClaferIG) [ref] had bugs related to references, so we were unable to generate instances. For the UML model, we first commented out the extensions we made to the specification and then we compiled it successfully in USE tool.

3.1 Evaluation Criteria

We decided on the set of criteria for the evaluation of both modeling languages based on observable characteristics of a model and the common design considerations that are taken into account when designing a modeling language. These criteria would also serve as the basis for evaluation in the future empirical study [8].

3.1.1 Model Size

The model size is strongly related to the ability of a modeling language to provide concise representation of the requirements of a certain domain. What is the total number of lines of code for each of the models? What is the total number of characters for each of the models?

3.1.2 Expressiveness

An important concern in evaluating a modeling language is whether the language is expressive enough. The expressive power of a language is usually determined by how far facts from the problem domain are expressed in an easy manner. Were certain requirements impossible to express? Were certain requirements difficult to express? This criterion is also very important when evolving models since increased knowledge often adds to the modeling difficulty.

3.1.3 Requirements Distribution

The ease with which a modeler creates or evolves a model depends on how parts of the model representing a requirement are distributed. As the context in which all information about a requirement is decreased, a modeler could easily see missing information about a requirement that is needed to be included.

How many parts for modeling a single requirement are distributed throughout the model?

3.1.4 Amount of Restructuring

Evolution of structural models occurs as a result of a series of modifications in the problem domain description of the system under test. The effect of changes could be as small as changing the name of attributes, or as significant as refactoring the whole model. In order to provide better support for model evolution, a modeling language should be able to accommodate modifications with minimum amount of restructuring. What types of restructuring occurred in our models? How many lines of code were added/deleted in a restructuring step?

3.1.5 Locality of Change

Since model evolution means that formal models are being refactored and refined continuously, it is important that changes, especially small ones, remain contained within a small context as much as possible to ease the process for the modeler. Do changes propagate throughout the entire model? Is there a way for global changes to be avoided?

3.1.6 Frequency of Errors

The types and number of errors frequently occurring in a model help in pointing out the weaknesses of a modeling language. What were the most common errors made when using the language? What was the frequency of these errors? Can we identify the reasons behind the occurrence of these errors? Can we find a way to avoid them?

3.1.7 Redundancy

To be able to achieve compactness in a formal model, one of the most important factors is avoiding redundancy. Was redundancy present in our models? Are there different types of redundancy? Can we identify causes of redundancy? Can we think of ways to avoid it?

3.1.8 Validation Mechanisms

Structural model validation is important to the modeling process since it investigates how closely does the model represent the required structure characteristics of the system. A model should provide means to support the integration of a validation mechanism within the same representation. Is validation possible without tool support? If yes, how useful is this mechanism in preventing/detecting mistakes? And how much do the generated instances resemble the abstracted concepts?

3.1.9 Speed of Modeling

The level of comfort and the ease with which a modeler is able to correctly model a requirement using a given language is reflected in the speed with which he/she models that requirement. When evolving models, it is important that the language supports the process in a fast manner to accommodate the rapidly changing nature of requirements elicitation. Which language allows for faster modeling?

3.1.10 Clarity and Understandability

Since we are concerned with evaluating Clafer and UML for their modeling capabilities and usability at the requirements stage, their primary focus would be providing humans with the ability to exchange ideas and thoughts about models. In this context, clarity and understandability are always important. A model is clear if it corresponds to recognition patterns of the user. Since recognition patterns vary among different users, this would be a subjective criterion and hence would be tested only in the empirical study to have a considerable number of subjects instead of just one. Understandability, on the other hand, could be measured through how far a developed model would help the users learn about the domain. Of course the familiarity with the concepts and notation of a language would be a factor influencing both clarity and understandability. In order to gather data about these criteria, one proposed idea is to give the subjects the developed domain models and a prepared set of questions about the domain. We would count the number of correct answers, the speed of answering these questions, and finally ask the subjects to respond to a questionnaire to collect some qualitative data such as the subjects' confidence level in their submitted answers.

4. RESULTS AND ANALYSIS

In this section we present the analysis performed on the developed models based on the previously defined evaluation criteria. The purpose of this analysis is to capture the differences between Clafer and UML in structural modeling, identify the strengths and weaknesses of each language which would finally help in the design of the empirical study.

4.1 Model Size

In comparing the model size for both languages we excluded namespaces, comments and we tried to use the same syntax for language constructs expressing the same concepts. For example, the concept **View** is presented using its name in both models. The total number of lines of code for Clafer –without the concrete clafers used for validation- is almost 60% of the corresponding textual UML model. The number of characters for the USE model is almost twice as much the corresponding Clafer model. Since the same number of requirements have been modeled in each of them, this measure indicates that a Clafer model has a more succinct representation than a textual UML one.

Table 1. Points of Comparison for the model sizes of both Clafer and Textual UML

Point of Comparison	Clafer	Textual UML
Total Number of Lines of Code	150	250
Total Number of Characters	4647	8456

4.2 Expressiveness

Clafer encourages the use of hierarchies/nesting to provide a concise notation for modeling. For example, if we look at the requirements for modeling the "metric" concept:

- *Requirement #3:* The planning processes are supported by key financial indicators (metrics) that include sales, markdown, turn, receipts, inventory, gross margin, and open-to-buy.
- *Requirement #18:* A measure is defined for a specific metric, UOM, and a plan version it belongs to.
- *Requirement #31:* MFP users can plan sales based on three classifications; regular, promotional and clearance sales.
- *Requirement #32:* Markdowns are classified into regular, promotional and permanent markdowns.

First, we consider expressiveness from the point of how much information a given scope of a model can convey to end users. Fig (1) shows a Clafer model fragment that represents the above requirements. The abstract clafer **metric** having an **xor** group cardinality to select among **sales**, **markdown**, **turn**, etc. Further requirements evolve the model where both **sales** and **markdown** could be further classified. This is simply supported by nesting those possibilities. For example, the nesting of **regularSales**, **promotionalSales** and **clearanceSales** implies that they are sub-categories of **sales.**

```
abstract metric
        xor metricName
                xor sales
                        regularSales
                        promotionalSales
                        clearanceSales
                xor markdown
                        regularMarkdown
                        promotionalMarkdown
                        permenantMarkdown
                turn
                receipts
                inventory
                grossMargin
                openToBuy
```

Figure 1: Modeling of the Metric Hierarchy in Clafer.

However, in UML shown in Fig (2) a decision to flatten the hierarchy and represent the possibilities as an enumeration was taken since the metric has only one value at a time. Although this way of modeling has fewer lines of code, we lose knowledge of the domain where **regularSales, promotionalSales**, and **clearanceSales** are in fact sub-categories of **Sales** and this makes it less expressive. If we decide to represent this hierarchy in UML, we will have to include OCL constraints similar to the one shown in Fig (3) for all types of sales and markdown. This adds to the length of the model, and reduces its readability at the same time.

```
enum Metric { regularSales, promotionalSales, clearanceSales,
      regularMarkdown, promotionalMarkdown, permenantMarkdown,
      turn, receipts, inventory, grossMargin, openToBuy}
```

Figure 2: UML modeling of Metric as an enumeration which leads to partial loss of knowledge.

```
context metric
inv CheckSales:
        self.regularSales.isDefined() implies
        self.metricName =#Sales
```

Figure 3: Alternative for Metric Modeling in UML using OCL constraints.

Expressiveness could also be considered as the ability to model certain requirements without any regard to whether it is done concisely or not. Difficulty in modeling requirements mainly occurs when trying to model system constraints. In Clafer, every nested clafer denotes a relation. Those relations however have to be expressed as associations or OCL constraints in UML. For example, consider the following requirements:

- *Requirement #19:* Measures are classified into reference and non-reference measures (historical ones).
- *Requirement #21:* A non-reference measure is meant to be edited by a specific role.
- *Requirement #22:* Reference measures can't be edited by any role.

Looking at Fig (4), the nesting of **editedBy** reference below **nonReferenceMeasure** clafer, implies the constraint (Req#22) that only non-reference measures could be edited.

```
xor measureType
        referenceMeasure
        nonReferenceMeasure
                editedBy -> planningRole ?
```

Figure 4: Nesting in Clafer indicates a relation.

In UML, this has to be modeled using an explicit OCL constraint as shown in Fig (5). This not only adds to the length of the model, but also the complexity level increases since now we have to define association roles and then use them in the constraint.

```
association RoleEditsMeasures between
Measure [1..*]          role EditedMeasure
PlanningRole[1]         role EditedBy
end

context Measure
inv EditedMeasureCondition:
        self.EditedBy.isDefined()   implies
        self.measureType =#NonReference
```

Figure 5: Explicit OCL constraint to represent a relation.

Attempting to access a collection using an OCL expression is certainly one of the examples showing an increased difficulty in modeling constraints using UML compared to Clafer. Consider the following requirements:

- *Requirement #12***:** MFPs follow workflows for creating /managing plans, and each workflow has one or more views.
- *Requirement #13:* There are views who are meant to be seen by a single specific role, and others that could be seen by all roles.
- *Requirement #16(d):* Waiting for Approval (Wa); A plan awaiting approval by the Middle-out role.
- *Requirement #33:* In the view, you choose to seed, approve, create, or review a plan.

The **View** has more than one **planningRole** associated with it depending on the type of **View.** A **planningRole** on the other hand can see more than one view. This means that the relation between **View** and **planningRole** –in a general sense- is many-to-many. In Clafer, we can distinguish which part of the **View** is associated with one **planningRole** instance, and which is associated with many instances. Fig (6) shows the constraint required by requirement 16; the constraint is nested which implies you have to be in **Approval** view, its syntax shows that we don't have to provide the full path name to fetch where **specificRole** is**.** This feature of Clafer is called contextual name lookup. It helps in keeping constraint expressions simpler, and save the modeler the effort of providing the full path view to a certain clafer.

```
xor viewType
        Approval
                [ specificRole.roleLevel = MiddleOut ]
```

Figure 6: Clafer Name Lookup Feature.

In order to model the same constraint in OCL, we have to introduce a new function called **exists().** Since we only have one

association between **View** and **planningRole** and it is many-to-many, we will be attempting to access a collection of instances of **planningRole** and select one **roleLevel** of type **MiddleOut**, thus a different way of handling the constraint is required which is shown in Fig (7). The OCL constraint is lengthier, and more complicated since you have to check whether a certain variable exists or not, check its value, and then proceed to the remaining part of the constraint.

```
inv ApprovalView:
self.Type =#Approval  implies
self.visibility =#SpecificRole and
self.planningRole -> exists (roleLevel =#MiddleOut)
```

Figure 7: OCL constraint structure which requires checking whether a variable is null or not before executing the rest of the expression.

4.3 Requirements Distribution

This measure could be evaluated based on the amount of information conveyed about a certain concept from reading a certain part of the model. Additional **View** requirements to the ones mentioned above:

- *Requirement #14:* Each view has one or more measures.
- *Requirement # 20:* All measures are visible by all roles.
- *Requirement #33:* In the view, you choose to seed, approve, create, or review a plan.

In Fig (8), we can see a Clafer model fragment for modeling the **View**. In about 21 consecutive lines, we can see all of the information related to the **View** from the explicitly defined requirements mentioned above, and from constraints implied from other requirements. UML captures the same requirements as shown in Figs (9,10), but in almost 45 non-consecutive lines.

```
abstract view
        viewBelongsTo -> workflow
        xor visibility
                specificRole -> planningRole
                        [ viewBelongsTo.workflowPlanningSeason.inSeasonPlanning =>
                          !specificRole.roleLevel = TopDown ]
                allRoles
                        roles -> planningRole * = planningRole
        measures -> Measure +
        planversion -> plan
        [ viewBelongsTo.workflowPlanningSeason.inSeasonPlanning =>
          !(planversion = WaitingForApproval || planversion = LastYear)]
        xor viewType
                Approval
                        [ specificRole.roleLevel = MiddleOut ]
                        [ planversion = WaitingForApproval ]
                        [ viewBelongsTo.workflowPlanningSeason.preSeasonPlanning]
                Seeding
                        [ viewBelongsTo.workflowPlanningSeason.preSeasonPlanning]
                        [ specificRole.roleLevel =TopDown || specificRole.roleLevel = MiddleOut]
                        [ planversion = WorkingPlan ]
                Review
                Planning
```

Figure 8: Fragment of the Clafer Model Fragment for the concept View.

In the UML model, we first declare the class **View**. As we write the attributes, we have to scroll up the model by about 120 lines in order to declare the enumerations then scroll down to the middle section of the model by about 70 lines to declare the necessary associations. Finally, we scroll down the model again by about 120 lines to write down the required OCL constraints –Fig 10-. Navigating through the UML model per requirement often occurs iteratively because you need to refer to the exact names of attributes, association role names in the constraints.

```
-----------------------------------------------------

enum ViewVisibility {SpecificRole, AllRoles}
enum ViewType {Approval, Seeding, Review, Planning}

-----------------------------------------------------

class View
attributes
        visibility: ViewVisibility
        Type: ViewType

end

-----------------------------------------------------

association RolesAndViews between
View [1..*]
PlanningRole [1..*]
end

association PlansAssociatedWithViews between
Plan[1..*]
View[1..*]
end

aggregation ViewHasMeasures between
view [1..*]
Measure [1..*]
end

composition WorkFlowHasViews between
workflow [1]
view[1..*]
end
```

Figure 9: Fragment of the UML Model showing the View class declaration, enumerations, and associations.

4.4 Amount of Restructuring

In order to compare the amount or type of restructuring needed by each language to support model evolution, we chose requirements for a random concept and we performed step-by-step Clafer and UML modeling. Consider the following requirements for the relation between **planningRole** and **Target**. Assume that the clafer/class **plan** already exists in both models.

- *Requirement #4*: There are three types of planning roles in MFPs; Top-down, Middle-out, and Bottom-up.

Clafer modeling added 5 lines of code to the model as shown in Fig (11). UML modeling also added 5 lines of code as shown in Fig (12), but the modeler has to navigate through one hundred lines of code between the enumeration and the class declaration.

```
context View
inv SpecificRoleVisibility:
self.workflow.workflowPlanningSeason =#InSeasonPlanning and
self.visibility =#SpecificRole implies
(not (self.planningRole ->exists (roleLevel=#TopDown)))

inv AllRolesVisibility:
self.visibility =#AllRoles implies  self.planningRole -> includesAll(self.planningRole)

inv viewSeasonPlans:
self.workflow.workflowPlanningSeason =#InSeasonPlanning implies
(not (self.plan -> exists(oclIsTypeOf(WaitingForApprovalPlan) or oclIsTypeOf(LastYearPlan))))

inv ApprovalView:
self.Type =#Approval  implies
self.visibility =#SpecificRole and
self.planningRole -> exists (roleLevel =#MiddleOut) and
self.workflow.workflowPlanningSeason =#PreSeasonPlanning and
self.plan -> exists(oclIsTypeOf(WaitingForApprovalPlan))


inv SeedingView:
self.Type =#Seeding  implies
self.visibility =#SpecificRole and
(self.planningRole -> exists (roleLevel =#TopDown) or
self.planningRole -> exists (roleLevel =#MiddleOut)) and
self.workflow.workflowPlanningSeason =#PreSeasonPlanning and
self.plan -> exists(oclIsTypeOf(WorkingPlan))
```

Figure 10: UML Model Fragment showing the View OCL constraints.

```
abstract planningRole
        xor roleLevel
                TopDown
                MiddleOut
                BottomUp
```

Figure 11: Clafer Modeling of Requirement 4.

```
enum PlanningRoleLevel { TopDown, MiddleOut, BottomUp}

class PlanningRole
attributes
roleLevel: PlanningRoleLevel

end
```

Figure 12: UML Modeling of Requirement 4.

- *Requirement #5:* Top-down roles are typically planning directors. They create the overall targets for the company and set top-down group level targets for the middle out role.

```
abstract planningRole
       xor roleLevel
              TopDown
                     createTopDownTargets -> Target +
              MiddleOut
                     receivesTopDownTargets -> Target +
              BottomUp

abstract Target
       createdBy -> planningRole
       [ createdBy.roleLevel=TopDown => publishedTo.roleLevel = MiddleOut ]

       publishedTo -> planningRole
```

Figure 13: Clafer Modeling of Requirement 5.

Clafer modeling added five lines and one constraint in the same part of the model. UML modeling added eight lines and one constraint scattered throughout the model; about 50 lines between the class declaration and the associations, and 50 other lines to write the constraints.

```
enum PlanningRoleLevel { TopDown, MiddleOut, BottomUp}
-----------------------------------------------
class PlanningRole
attributes
roleLevel: PlanningRoleLevel

end

class Target
attributes

end
-----------------------------------------------

association TargetCreatedBy between
Target [1..*]          role  createdTarget
PlanningRole [1]       role  TargetCreator
end

association TargetPublishedTo between
Target [1..*]          role  publishedTarget
PlanningRole [1]       role  TargetReceiver
end-----------------------------------------------

context Target
inv TargetsPublishedTo:
        self.TargetCreator.roleLevel = #TopDown implies
        self.TargetReceiver.roleLevel = #MiddleOut
```

Figure 14: UML Modeling of Requirement 5.

- *Requirement #6:* Middle-out roles are typically planning managers. They create middle-out targets.

Clafer modeling for the above requirement added only one line while nothing was changed in the UML model, since the association between the **planningRole** and **Target** has already been created.

```
abstract planningRole
        xor roleLevel
                TopDown
                        createTopDownTargets -> Target +
                MiddleOut
                        receivesTopDownTargets -> Target +
                        createMiddleOutTargets -> Target +
                BottomUp

abstract Target
        createdBy -> planningRole
        [ createdBy.roleLevel=TopDown =>
                publishedTo.roleLevel = MiddleOut ]

        publishedTo -> planningRole
```

Figure 15: Clafer Modeling of Requirement 6.

- *Requirement #7:* Bottom-up roles are typically merchandise planners. They create the Op (Original Plan) and create the Cp (Current Plan) plans for approval by the middle out role.
- *Requirement #8:* The targets are published by superior levels to the subsequent level: top down passes to middle out, and middle out passes to bottom up.

```
abstract planningRole
        xor roleLevel
                TopDown
                        createTopDownTargets -> Target +
                MiddleOut
                        receivesTopDownTargets -> Target +
                        createMiddleOutTargets -> Target +
                BottomUp
                        receivesMiddleOutTargets -> Target +
                        createPlans -> plan +

abstract Target
        createdBy -> planningRole
        [ createdBy.roleLevel.TopDown =>
                publishedTo.roleLevel.MiddleOut ]
        [ createdBy.roleLevel.MiddleOut =>
                publishedTo.roleLevel.BottomUp ]

        publishedTo -> planningRole
```

Figure 16: Clafer Modeling of Requirements 7, 8.

Clafer modeling for requirements 7 and 8 added two lines and one constraint. However, we had to navigate to **plan** to add a reference to a **planningRole**. Taking care of inverses manually is a disadvantage due to the absence of associations in Clafer. Although the navigation through the model to add those inverses is certainly through less lines of code than UML (only 20 lines), it adds to the types of restructuring needed for evolving models and negatively affects the locality of change. UML modeling for those requirements added more lines of code, and two restructuring types; associations and constraints.

```
association RoleCreatesPlans between
PlanningRole [1..*]
Plan [1..*]
end

context Target
   inv TargetsCreatedBy:
        self.TargetCreator.roleLevel = #TopDown or
        self.TargetCreator.roleLevel = #MiddleOut

inv TargetsPublishedTo:
        self.TargetCreator.roleLevel = #TopDown implies
        self.TargetReceiver.roleLevel = #MiddleOut or
        self.TargetCreator.roleLevel = #MiddleOut implies
        self.TargetReceiver.roleLevel = #BottomUp


inv TargetsNotPublishedTo:
        self.TargetReceiver.roleLevel = #MiddleOut or
        self.TargetReceiver.roleLevel = #BottomUp
```

Figure 17: Added UML parts to model Requirements 7, 8.

4.5 Locality of Change

In this section, we will explore some evolution examples that needed restructuring in both Clafer and UML models to be able to analyze the locality of change associated with each modeling language. The example discussed in the previous section is not only related to the amount of restructuring, but also explores the locality of change. We saw how requirements evolution is usually localized in Clafer, while causing changes throughout the entire model of UML.

In the final iteration of Clafer modeling, we extracted the plan versions from nested clafers into separate abstract ones all inheriting from the abstract clafer **plan**. We extracted clafers because we found that we added constraints in different parts of the model to ensure that we are talking about a specific plan version. We applied inheritance since all plan versions share common features such as the plan's **versionAbbreviation**, and the **planningRole(s)** which created it. Figure (19) shows two Clafer model fragments with **planVersion** constraints at the top part, and the bottom part after the application of inheritance. The extraction of all plan versions into abstract clafers was done directly underneath the **plan** clafer in the same part of the model. Figure (18) also shows the **Target** model fragment before and after the application of inheritance. Effectively, two constraints were removed, and three were modified inside the **View** clafer which is almost 40 lines below the plan clafers.

```
abstract Target
        belongsTo -> planVersion
        [belongsTo.planVersionName.TargetPlan]
-----------------------------------------------
abstract Target
        belongsTo -> Targetplan
```

Figure 18: Clafer Model Fragment for Target before and after inheritance showing less number of constraints.

```
abstract plan
       xor planVersion
              WorkingPlan
                     [ versionAbbreviation = Wp ]
                     seeded?
                            seedingSource -> plan
                                   [ seedingSource.planVersion.LastYear]
--------------------------------------------------------------------------
abstract WorkingPlan: plan
       [ versionAbbreviation = Wp ]
       seeded?
              seedingSource -> LastYear
```

Figure 19: Clafer Model Fragments before and after inheritance.

4.6 Frequency of Errors

One of the main principles on which this study is based, is to create and evolve the models without any tool support in order to analyze the models based on the modeling language features only. The tools for both Clafer and USE were used eventually to verify the models, and to give insight about the types of errors that were made and their frequency. For the Clafer models there were three types of errors, listed below in order from highest frequency of occurrence to the lowest.

- *Missing References:* Modeling using references nested underneath clafers means that we have to take care of inverses ourselves. The example shown in Fig (20) demonstrates this concept. When we want to model the requirement that a **View** has one or more measures, we add a reference to the **measure** with clafer cardinalities inside **View,** but we also have to navigate to the **measure** clafer, and add a reference to **View.** The highest frequency of mistakes occurred due to forgetting to include these inverses.

```
abstract view
        measures -> Measure +

abstract Measure
        belongsTo -> view+
```

Figure 18: References and Inverses in Clafer.

- *Missing Constraints:* This error occurred when the same information is related to more than one concept in the model. Consider the **WorkingPlan** example shown in Figs (21, 22) with the following requirements:
  - *Requirement #27:* In the pre-season process, the working plan can be seeded with Last Year (Ly) data – This represents the seeding source- to create a demand curve on which to spread the new plan's initial targets. Or, you can instead choose to not seed the plan, which allows you to create a plan that is not influenced by last year's performance.
  - *Requirement #28:* If you decide to seed the plan, you should keep track of the last seeding date
  - *Requirement #29:* When seeding a plan, you choose which information to seed. You can seed certain levels

of each hierarchy (product, calendar, location) or all levels.

```
abstract WorkingPlan: plan
        [ versionAbbreviation = Wp ]
        seeded?
                seedingSource -> LastYear
                lastSeedingDate: string
                seededWithThoseInstances -> HierarchyLevelInstance *
```

Figure 19: Working Plan Information.

In Fig (21), we capture all of the WorkingPlan requirements, but within the context of the **WorkingPlan** only. However, later requirements show that there is a **Seeding** view which is in concept related to the seeding process of the working plan, thus it should have the same constraints as shown in Fig (22).

```
Seeding
        [ viewBelongsTo.workflowPlanningSeason.preSeasonPlanning]
        [ planVersion = WorkingPlan]
```

Figure 20: Working Plan Inverses.

This error could also be related to the first one since sometimes constraints are not only modeled in the context of the current clafer, but apply to inverses as well. Inverses were often missed in the preliminary versions of the model, but they were corrected in the following iterations.

- *Syntax Errors:* The third error type that was encountered was a syntax error in selecting a grouped clafer. The mistake was a result of misunderstanding that we use the "=" operator when trying to specify any value in a concrete clafer as shown in Fig (21). However, the "=" operator should only be used when trying to assign a value to an enumeration or a reference, and the dot operator should be used when choosing among grouped clafers as shown in Fig (22).

```
abstract plan
        xor retailChannels
                store
                internet
                catalog

abstract originalPlan: plan

originalPlan2012: originalPlan
        [ retailChannels = store ]
```

Figure 21: Syntax error for selecting the retailChannels grouped clafer.

```
abstract plan
        xor retailChannels
                store
                internet
                catalog

abstract originalPlan: plan

originalPlan2012: originalPlan
        [ retailChannels.store ]
```

Figure 22: Correct Syntax for selecting a grouped clafer.

For the UML models, most of the errors were in modeling OCL constraints categorized into the following types listed in a descending order of frequency of occurrence:

- *Choosing the correct constructs for OCL constraints:* The highest number of errors occurred due to difficulty in writing OCL constraints. The most difficult constraints were that of the **View**. Selecting among a collection in OCL requires familiarity with the concepts of existential quantification or universal quantification. These concepts are also present in Clafer, but they are hidden due to the fact that constraints are always placed in context. Applying those concepts, searching for the correct sequence of using their operators, and including all dependencies was the most difficult part of the whole modeling experience. Examples are shown in Fig (23).

```
-- Approval View Constraints
   inv ApprovalView:
        self.Type =#Approval  implies
        self.visibility =#SpecificRole and
        self.planningRole -> exists (roleLevel =#MiddleOut) and
        self.workflow.workflowPlanningSeason =#PreSeasonPlanning and
        self.plan -> exists(oclIsTypeOf(WaitingForApprovalPlan))


-- Seeding View Constraints
   inv SeedingView:
        self.Type =#Seeding  implies
        self.visibility =#SpecificRole and
        (self.planningRole -> exists (roleLevel =#TopDown) or
        self.planningRole -> exists (roleLevel =#MiddleOut)) and
        self.workflow.workflowPlanningSeason =#PreSeasonPlanning and
        self.plan -> exists(oclIsTypeOf(WorkingPlan))
```

Figure 23: OCL Constraints.

- *Association Mistakes:* Trying to model the **Workflow** concept in UML was a bit confusing. Current workflow could be preceded or followed by other workflows. First, we modeled using references like Clafer, but then discovered we don't need the extra added constraints, and it is better to use associations. Second, we modeled using a single association, but also proved not to work since the proceeding workflow of the current instance is different than the following instance. Finally, we found that the latest version of USE supports the use of role names on associations in constraints' declarations. This was the final design.
- *Syntax Errors:* The syntax for accessing an enumeration in OCL is different from regular UML code, and the USE documentation didn't provide examples. However, a paper published by the developers of USE [5] had an example on using enumerations, and the mistakes were resolved. This mistake was repeated about 6 times before the correct syntax was found.

4.7 Redundancy

A modeling language should be designed so that redundancy is minimized as much as possible or entirely eliminated. Although the Clafer model is much more concise (150 lines) compared to the UML model (250 lines), information redundancy was present more in Clafer than in UML. This is due to the fact that all relationships among Clafers are done using references, and modeling of inverses. This sometimes introduces redundancy in constraints. Figure (24) shows this type of redundancy when modeling the requirement that **BottomUp** roles can't create targets. Since we are using references to establish a relation between **planningRole** and **Target**, inverses had to be taken care of. This resulted in modeling the constraint twice. This kind of redundancy is not present in UML due to the presence of associations which automatically ensures the inverse constraints in both directions.

```
abstract planningRole
        roleLevel -> PlanningRoleLevel

        roleCreatedTheFollowingTargets -> Target *
        [ roleLevel=BottomUp => no roleCreatedTheFollowingTargets ]


abstract Target
        createdBy -> planningRole
        [! createdBy.roleLevel = BottomUp]
```

Figure 24: Redundancy in Clafer Models.

4.8 Validation Mechanisms

Both Clafer and UML in general support validation mechanisms. Clafer provides concrete clafers which are instances of abstract clafers representing real world examples. UML supports Object Diagrams where objects are instances of the classes in Class Diagrams, but this is only in Graphical UML. No such validation mechanism exists for the textual notation. Consider the example shown in Fig (25) where we represent an instance of the abstract clafer **Target**. The instance is a means to help the modeler ensure that he/she satisfies the constraints of Target, include all mandatory nested Clafers. It is also useful in pointing missing references or constraints when you write a real example below the abstract part. For example if we try to write an instance of Target and mention that it is created by a certain instance of planning role, we can look at the abstract clafer Target and find that a reference **createdBy** might be missing.

```
abstract Target
        belongsTo -> Targetplan

        createdBy -> planningRole
        [! createdBy.roleLevel = BottomUp]

        publishedTo -> planningRole
        [ createdBy.roleLevel=TopDown => publishedTo.roleLevel = MiddleOut ]
        [ createdBy.roleLevel=MiddleOut => publishedTo.roleLevel = BottomUp ]


//********** Concrete Clafers ********************************************

TopDownSalesTargets2012: Target
        [  belongsTo = TargetPlan2012
           createdBy = ExecutiveManager
           publishedTo = PlanningDirector ]
```

Figure 25: Concrete Clafer for Target.

4.9 Speed of Modeling

Random requirements were chosen during the modeling process to compare the time needed to model using both languages. We start with the **Workflow** requirements mentioned below:

- *Requirement #23:* If you are doing pre-season planning, then it can't be proceeded by in-season planning.
- *Requirement #24:* Once you are in-season planning, you can't return to the pre-season planning stage.

The time taken to model these requirements in Clafer took about 2 hours. This long time was taken because requirements 23, and 24 were a bit ambiguous in the sense of which abstract clafer they belonged to.

```
abstract workflow
        belongsToThisSystem -> MFPsystem

        xor workflowPlanningSeason
                preSeasonPlanning
                inSeasonPlanning
```

Figure 26: Clafer Modeling for the planning processes.

We started with making a separate clafer for the planning process, and added an **XOR** group cardinality with the possible types; as pre-season and in-season planning. A reference to the planning process clafer was added inside the **Workflow** clafer with attribute cardinality 2. After reading the whole requirements, we found that the planning process is only related to the workflows, so we deleted the planning process clafer, and added the information as nested group cardinality inside the Clafer model as shown in Fig (26). UML modeling of these requirements took less time although it involved more lines of code. This is due to the fact that we had prior knowledge of the problem domain, and modeled the UML model similarly.

The other requirement we chose was that of the hierarchy levels shown below:

- *Requirement #11:* Users may edit data at many levels of each hierarchy. If the data is modified at an aggregate level, the modifications should be distributed to the lower levels

beneath it. This process is called spreading. If data is modified at a level that has a higher level above it, the data changes are reflected in those higher levels. This is known as aggregation.

For Clafer modeling, it took about an hour to model the requirement as shown in Fig (27). It took a couple of minutes to add to references to **HierarchyLevel** instances, but it took some time to figure how to write the constraint and whether we needed to make **spreadsToLowerLevel**, and **aggregatesToHigherLevel** optional clafers. Concrete clafers were included to help making this decision; year didn't aggregate to any level in the calendar hierarchy, and week didn't spread to anything. The concrete clafer helped pointing a missing reference: **levelBelongsToThis**. The level had to belong to a certain type of hierarchy; year belonged to Calendar Hierarchy.

UML modeling of requirement #11 took more time –about an hour and 30 minutes- although the same requirement was first modeled in Clafer and problem domain knowledge should have influenced the speed. Modeling the requirement ended up in more lines of code (16 lines) –shown in Fig (28)- compared to Clafer (4 lines). Most of the time spent was trying to write a correct syntax for the OCL constraints.

```
abstract HierarchyLevel
        levelBelongsToThis -> Hierarchy
        spreadsToLowerLevel -> HierarchyLevel ?
        aggregatesToHigherLevel -> HierarchyLevel ?
        [ all disj d1; d2 : HierarchyLevel |
        (d2 = d1.aggregatesToHigherLevel <=> d1 = d2.spreadsToLowerLevel) ]
```

Figure 27: Clafer Modeling of Requirement 11.

```
class HierarchyLevel
attributes

end

association LevelSpreadsTo between
HierarchyLevel[1]       role   base
HierarchyLevel[0..1]    role   lowerLevel
end

association LevelAggregatesTo between
HierarchyLevel[1]       role   currentLevel
HierarchyLevel[0..1]    role   higherLevel
end

context HierarchyLevel
inv AggregatesTo:
        HierarchyLevel.allInstances()->
        forAll(l1, l2| l1.higherLevel=l2 implies l2.currentLevel=l1)

inv SpreadsTo:
        HierarchyLevel.allInstances()->
        forAll(l1, l2| l1.lowerLevel=l2 implies l2.base=l1)
```

Figure 28: UML Modeling of Requirement 11.

5. DISCUSSION

In this section, we attempt to give preliminary answers to the proposed research questions as a result of the performed pilot study. These answers are intended to be used to formulate a clear hypothesis for the empirical study to be performed.

5.1 Evaluation

5.1.1 Model Size

What are the total number of lines of code for each of the models? What are the total number of characters for each of the models?

Clafer model is about 150 lines and 4000 characters while the UML model is about 250 lines and 8000 characters, which shows that Clafer has a more succinct representation the textual UML (USE specification).

5.1.2 Expressiveness

Were certain requirements impossible to express? Were certain requirements difficult to express?

All requirements could be expressed using both modeling languages. A tradeoff between complexity and length for UML was observed when trying to model hierarchies. Group cardinality was useful in Clafer, and associations were useful in UML. Name lookup feature in Clafer shows great potential and needs further investigation.

Although both languages were able to model all requirements, there is a significant difference between Clafer and UML/OCL modeling. Clafer is developed based on unifying concepts; clafers and constraints. UML, on the other hand, is based on having several concepts; classes, associations, and constraints. It was much easier for me to model constraints in Clafer since I only needed to learn a few keywords and how to place constraints in the correct context. Although I was previously familiar with UML, it was much more difficult to learn and use OCL. I think this difference would be better tested in the empirical study since it would definitely affect the results of familiarity of notation as well as clarity/understandability criteria.

5.1.3 Requirements Distribution

How many parts for modeling a single requirement are distributed throughout the model?

A Clafer model groups information related to a concept in a single part of the model except for inverses if applicable. A USE model however separates different parts of the same requirement throughout the model by dividing a model into separate sections for enumerations, class definitions, association definitions, and constraints. This is a significant disadvantage for the modeler from a traceability point of view. If I wanted to model a certain requirement, I had to write an external note whether I needed to include only a class definition or maybe also an association or a constraint so that I wouldn't forget to include any required part from having to navigate throughout the model. This criterion would affect that of clarity/understandability in the empirical study. Having the requirements distributed across different parts of the model might confuse the subjects or makes it more difficult for them to understand the domain.

5.1.4 Amount of Restructuring

What type of restructuring occurred in our models? How many lines of code were added/deleted in a restructuring step?

For Clafer modeling, restructuring occurred in several forms such as extracting abstract clafers from nested ones to make them separately instantiable, using nesting/hierarchies to denote a relation, using local constraints, adding inverses, and finally changing a group cardinality to inheritance. For UML, most restructuring took place in the form of adding/deleting classes, associations and constraints. Clafer modeling involved more

restructuring types, but fewer lines of code per a restructuring step.

5.1.5 Locality of Change

Do changes propagate throughout the entire model? Is there a way for global changes to be avoided?

For Clafer, changes to a requirement only propagate in case they need to be reflected back at inverse constraints. This could be avoided if associations were present. For UML, changes always propagate through the model due to the USE model being divided into separate sections for enumerations, classes, associations and constraints.

5.1.6 Frequency of Errors

What were the most common errors made when using the language? What was the frequency of these errors? Can we identify the reasons behind the occurrence of these errors? Can we find a way to avoid them?

For Clafer, the types of errors in order of frequency were missing references, missing constraints and syntax errors. The first two errors occurred due to inverses, and can be avoided by introducing associations. For UML, most errors occurred in attempting to write OCL constraints and searching for the correct syntax. This could be minimized by giving proper training materials on OCL before using it.

5.1.7 Redundancy

Was redundancy present in our models? Are there different types of redundancy? Can we identify causes of redundancy? Can we think of ways to avoid it?

Redundancy was present in both models. For Clafer, the redundancy source was taking care of inverses due to the absence of associations. Proposals to include associations are currently being investigated. For the UML model, redundancy was present in having to explicitly specify all conditions by means of OCL constraints.

5.1.8 Validation Mechanisms

Is validation possible without tool support? If yes, how useful is this mechanism in preventing/detecting mistakes? And how much do the generated instances resemble the abstracted concepts?

Concrete clafers represent a validation mechanism that could be integrated within a Clafer model without the need for tool support. Concrete clafers help in detecting missing references or information when placed below the abstract clafers they represent. Concrete clafers resemble the abstract clafers in their structure except for having one nesting level. A textual UML model on the other hand doesn't support a validation mechanism without tool support.

5.1.9 Speed of Modeling

Which language allows for fastest modeling?

The answer to this question can't be determined at this stage since only one person was modeling, object learning effect might affect the results, and we don't have recording of the time taken to model each requirement; only 3 requirements were selected.

5.2 Threats to Validity

In this section, we discuss the threats to the validity of our results. These threats were the main reason why our results are only preliminary, and why we need to pursue the empirical study. Possible threats include that modeling was performed by a single person, previous knowledge of UML class diagrams, exposure to different training and reading materials for the modeling languages, object learning effect since UML modeling was done after the completion of Clafer models, and finally easy access to Clafer developers.

6. CONCLUSIONS

In this pilot study, our research was focused on getting preliminary results to the defined research questions. Our conclusions are mainly classified into the following points; better knowledge of both modeling languages, common model evolution features, preliminary ideas about the strengths and weaknesses of each language with respect to their support for model evolution, and finally lessons learnt about how we would pursue the experimental design. Since we explored most of these points in the discussion section, this section will focus on the lessons learnt.

During the model analysis phase, we always brainstormed ideas about extensions to the models that would give insight into what kind of features would better highlight the advantages of each language with regard to model evolution support, and make good comparison points. We need to explore some features before we go into the experimental design such as local and global constraints in Clafer versus UML, using different group cardinalities such as (**or**, **mux)** in Clafer and how these concepts could be represented in textual UML, and finally using Clafer's name lookup feature in writing constraints within deep hierarchies, how we would restructure those constraints if the hierarchies change, and what are the corresponding alternatives in UML.

Finally, some important experimental design considerations should also be taken into account such as:

1. Subjects should only model the problem using one modeling language to avoid the object learning effect on the data gathered.
2. Randomization of the domain's requirements when giving them to the subjects should be kept to avoid any biased results.
3. Subjects will be asked to commit a version of their model after finishing each requirement to collect detailed data about the amount of restructuring, locality of change per evolution step, and speed of modeling
4. Proper training materials covering each language constructs that would be used should be prepared and handed off with an example before the experiment is performed.
5. Subjects should answer a background questionnaire regarding their modeling knowledge and experience.
6. Models should be developed using plain text editors to avoid the effect of any tool support on the gathered data.
7. Additional evaluation criteria could be used such as the number of different language constructs used, familiarity of notation which could be measured based on comparing the number of questions raised by each subject for each of the languages, and finally the understandability of each modeling language. This could be measured by giving the developed models to the subjects with a set of prepared questions about the domain. The data gathered would be in the form of the number of correct answers, speed of answering the questions, and a final questionnaire given to the subjects

to collect qualitative data such as each subject's confidence level in his/her answers.

8. Mistakes should be tracked to give insight on what kind of tool support is needed for model evolution.

7. RELATED WORK

The idea of structural domain modeling is not new. Although there isn't a large body of related work in this area to serve the requirements stage, UML/OCL has been compared to another lightweight structural modeling language called Alloy [8]. The given comparison is not supported by a specific modeling experience/example provided in the paper where the reader can see the mentioned differences. The paper discusses several differences based on published work for both modeling languages.

The first comparison point was the complexity differences between OCL and Alloy in handling constraints. The author argues that OCL is more complicated since it is applied in a context which includes subclasses, parametric polymorphism, operator overloading, and multiple inheritance. The second point was accuracy differences. The author claims that OCL has a lower accuracy due to the fact that it uses operations to describe some constraints. An operation may go into an infinite loop, be undefined, or cause problems when applied to several classes which have an inheritance relationship. The final point of comparison; expression differences was the only point supported by small examples. The author claims that although UML is generally more expressive than Alloy since it has more data types, more ways of describing system architecture, and problem domain modeling, Alloy has more powerful expressions in describing relations. For example, Alloy has a transitive closure operator which is not present UML. We are not aware of any other work done in this area.

8. ACKNOWLEDGMENTS

I would like to thank Joanne Atlee, Krzysztof Czarnecki, Michal Antkiewicz, and Kacper Bak for their valuable comments and support.

9. REFERENCES

[1] F Garcia, Nelio Cacho, Claudio Sant'Anna, Sergio Soares, Paulo Borba, Uira Kulesza and Awais Rashid 2007, *On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. European Conference on Object-Oriented Programming*, Berlin, Germany.

[2] Helen C. Purchase, Linda Colpoys, Matthew McGill, David Carrington and Carol Britton, 2001. *UML class diagram syntax: an empirical study of comprehension.* In the Australian Symposium on Information Visualization (Sydney, Australia, December 2001).

[3] K. Bak, K. Czarnecki, and A. Wasowski. *Feature and Class Models in Clafer: Mixed, Specialized, and Coupled.* Technical Report CS-2010-10, David R. Cheriton School of Computer Science, University of Waterloo, 2010.

[4] Mcintosh, P., Hamilton, M., and Schyndel, R. *X3D-UML: 3D UML State Machine Diagrams.* In *Proceedings of the 11th International Conference on Model Driven Engineering Languages and Systems (MoDELS)*, LNCS 5301, pp. 264-279, 2008.

[5] Martin Gogolla, Fabian Büttner and Mark Richters, 2005. *A UML-based specification environment for validating UML and OCL*, EADS Transportation, University of Bremen, Bremen , Germany.

[6] Martin Gogolla. Model Development in the UML-based Specification Environment (USE). In Ed Brinksma, David Harel, Angelika Mader, Perdita Stevens, and Roel Wieringa, editors, *Methods for Modelling Software Systems (MMOSS)*. IBFI, Schloss Dagstuhl, Germany, 2007. Dagstuhl Seminar Proceedings 06351. 3 pages.

[7] Oracle Retail Merchandise Financial Planning Retail, User Guide, Release 13.0.2, December 2008.

[8] Yujing He. *Comparison of the modeling languages Alloy and UML*, Department of Computer Science, Portland State University,USA