

# An Empirical Study of Specification by Example in a Software Engineering Tool

Hazem Qattous

Department of Computing Science  
Sir Alwyn Williams Building  
University of Glasgow, G12 8QQ, UK  
+44 141 330 0916  
hqattous@dcs.gla.ac.uk

Philip Gray

Department of Computing Science  
Sir Alwyn Williams Building  
University of Glasgow, G12 8QQ, UK  
+44 141 330 4933  
pdg@dcs.gla.ac.uk

Ray Welland

Department of Computing Science  
Sir Alwyn Williams Building  
University of Glasgow, G12 8QQ, UK  
+44 141 330 4968  
ray@dcs.gla.ac.uk

## ABSTRACT

Meta-CASE tools offer CASE tool specialisation by enabling a designer to specify a tool which is then generated automatically. Constraints are often used in such meta-CASE tools for governing the syntax and semantics of model elements and the values of their attributes. However, the constraint definition process is complex, time-consuming and error-prone. This paper presents an empirical study of the use of Specification by Example (SBE), based on the well-known notion of Programming by Example (PBE), as a user-computer interactive technique for such constraint specification. Two constraint specification techniques have been implemented in a meta-CASE tool a wizard that represents a conventional form-filling technique and an SBE technique that depends on the user providing one or more examples and the system inferring a list of possible intended constraints. The empirical study compared the wizard and SBE with respect to constraint definition correctness, task completion time, and user satisfaction. Two common modelling diagrams have been used, a State Transition Diagram and a Use Case Diagram. Results suggest that SBE is superior to the wizard in terms of measured criteria described above. Observations on the interaction of users with the system and opinions of participants are also presented.

## Categories and Subject Descriptors

D.2.2 [Software Engineering]: Design Tools and Techniques – Computer Aided Software Engineering (CASE).

## General Terms

Design, Experimentation, Human Factors, and Languages.

## Author Keywords

Specification by Example, Programming By Example, user study, example-based interface, Meta-CASE tools, Domain Specific Language.

## 1. Introduction

CASE tools are helpful to software engineers in increasing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ESEM'10, September 16–17, 2010, Bolzano-Bozen, Italy.

Copyright 2010 ACM 978-1-4503-0039-01/10/09...\$10.00.

productivity and shortening development time. One of the main advantages of CASE tools is their potential domain specificity. However, because of their cost, domain specific environments are not widely or commercially available [5]. Meta-CASE tools solve this problem by their ability to specify and generate domain specific CASE tools including software engineering design diagram editors, the experimental domain of this paper. A meta-CASE tool specifies a diagram editor by enabling a designer to specify the modelling language itself through a meta-modelling process. This process includes the specification of the diagramming language elements using a meta-model, and the specification of the operations and the constraints on these elements [14]. This process is complicated, needs time, and, in the case of several tools, depends on experts to do the specification [3].

In the field of meta-CASE tools, most research is directed towards enhancing the meta-models to improve the meta-modelling process. However, constraints, which play an important role in the CASE tool configuration and the meta-modelling process, attract little research attention [8]. Meta-CASE tools depend on constraints to define the syntax and semantics of the desired modelling language. Constraints are used to define the relations between the diagram elements and their relation to the diagram itself. In the context of diagram editors, constraints are considered as signs to guide architects to a good design solution. They are rules that limit the available alternatives to achieve a task, and enforce conformity with a specific design process methodology [9, 17, 20]. It is believed that improving the meta-modelling process can be achieved through enhancing and simplifying the constraint definition process.

In the context of meta-CASE tools, constraint definition is performed using a constraint programming language such as in “Metaview” [4] and “MetaBuilder” [3]. Consequently, such a tool user must learn a constraint programming language to accomplish the job. Several studies have attempted to reduce the difficulty of constraint definition within meta-CASE tools. In place of direct entry, a form-filling technique has been used such as in “MetaEdit+” [10]. A visual programming language has also been used in the meta-CASE tool Pounamu [12]. These examples indicate that constraint definition is not easy and some approaches have been offered to solve its complexity.

The aim of this research, of which this study is a part, is to improve the process of constraint definition as part of domain specific CASE tool specification in a meta-CASE tool. In particular, the work focuses on facilitating and simplifying the constraint definition process using Specification by Example

(SBE) technique which based on the notion of well-known technique programming by example (PBE). This technique has not been used before in the context of meta-CASE tools.

This paper presents an initial study to investigate the usefulness of (SBE) in the constraint definition domain. The next section describes an experimental meta-CASE tool, Diagram Editor Constraints System (DECS), that has been developed and used as a domain for the experiments. This is followed by a discussion of specification by example and the way this has been implemented in DECS. An empirical study is then presented, comparing constraint definition by example with a convention wizard specification method. The paper concludes with a discussion of the results and proposed future work.

## 2. Diagram Editor Constraints System (DECS)

DECS is an Eclipse-based meta-CASE experimental prototype designed to explore PBE as a software engineering constraint specification technique, called in this paper “Specification by Example” (SBE). Figure (1) shows the architecture of DECS. It generates constraint-based domain specific diagram editors specified by the editor designer. The generated diagram editors are inherited from the Graphical Editing Framework (GEF) and the Eclipse Modelling Framework (EMF) plug ins.

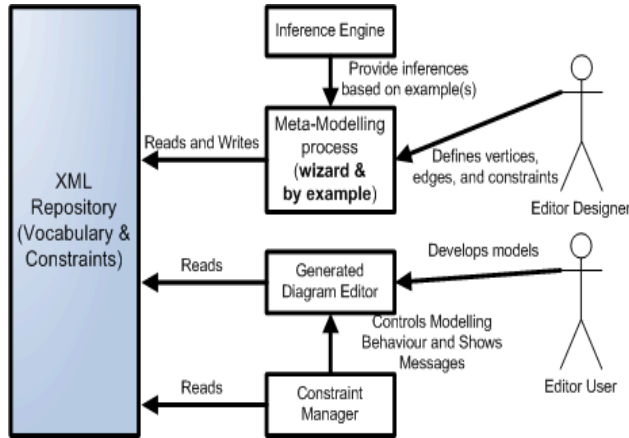


Figure 1: DECS Architecture

The CASE tool user (editor user in figure 1) uses the generated environment to develop system diagrams and models (e.g. UML Use Case Diagram or State Transition Diagram).

The following example clarifies the work of DECS. The editor designer in figure (1) wants to specify a State Transition diagram editor. S/he defines the diagram elements (vertices and edges). Vertices of the diagram include Start State, Non Terminal State, and End State while edges include Transition. After elements definition complete, s/he defines the syntax and semantics of the diagram using constraints. Using one of the constraint definition techniques available in DECS, the editor designer defines a set of constraints. One of these constraints, as an example, is “*it is not allowed to have more than one Start State in the diagram*”. After finishing his/her work, DECS generates a State Transition diagram editor that allows the editor user, in figure (1), to develop State Transition diagram models using the generated editor. If the user at anytime of the modelling process violates one of the

defined constraints, the editor behaves based on the type of the constraint [19]. In this example, if the user tries to draw a diagram with two Start States, the system detects a violation of a constraint, deletes the last added Start State, and shows a message to the user explaining the constraint. Figure (2) shows the generated editor and the constraint enforcement.

The constraint manager figure 1, at the CASE tool level, is responsible for constraint detection. Each time the user updates the diagram model, the constraint manager is consulted. If it detects a violation it either warns or prevents the user from violating a constraint. For further details about DECS refer to [19].

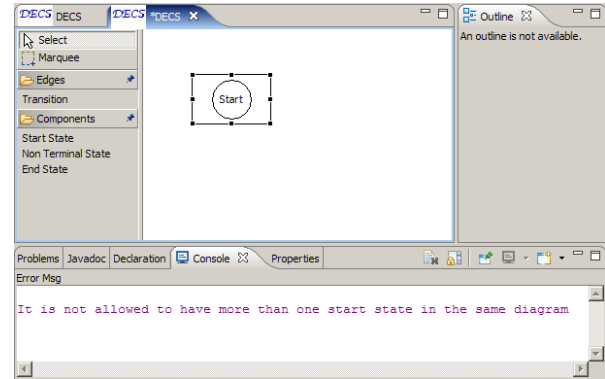


Figure 2: The error message “It is not allowed to have more than one start state in the same diagram” indicating violation of the constraint when the user tries to add the second start state.

## 3. Constraint Language

DECS depends on a simple XML based constraint language. Constraints in this language are classified into three types, vertex, edge, and graph constraints [19]. Each constraint is represented as an XML file with a set of properties and assigned values. Some of these values can be the URIs of other constraints as references. This ability to encapsulate constraints within each other makes it possible to construct complex constraints flexibly. It also allows the reuse of previously defined constraints which reduces the effort required to define a constraint. A similar feature has been introduced by [12] to express complex behaviour. The language can also express logical relations between referenced constraints such as logical “AND” and “OR”.

## 4. Programming By Example (PBE)

According to [15], PBE is the technique of presenting examples of data and values to the system, from which it can generalise these values to generate a program. The objective of PBE is to make programming an easier task and available for non-programmers. The technique depends on providing examples of the required program to the system. The system then infers the program by generalising the examples.

PBE has been applied in a number of different contexts such as “DocWizards” [18] for generating documentation and in mapping between a state and its associated actions in robotic applications [2]. It has also been applied to constraint specification by several researchers. Myers [15] used PBE in “Peridot” to infer

constraints between different graphical user interface components. Kurlander & Feiner [11] used the same technique to infer geometric constraints between vertices in a system called “Chimera”. We call the application of programming by example to a model specification “Specification by Example” (SBE) and will use that term hereafter.

## 5. Constraint Definition

In DECS, two constraint definition techniques have been implemented: form-filling represented by a “wizard” and Specification by Example (SBE) based on PBE. Both techniques are discussed in detail in the following two sections.

### 5.1 Constraint Definition by Wizard

DECS includes a wizard-based constraint definition facility as a basis for comparison with “by example” techniques. Many modern CASE tools use a form-filling technique for constraint definition as in MetaEdit+ [14], CASEMaker [15] and MetaBuilder [5]. This technique is also widely used in different contexts and almost all computer users are familiar with it. Finally, the property-value format of the XML based constraint language used in DECS makes the form-filling as ideal manual technique of specifying a constraint. However, because the number of constraint properties is large, it is more feasible to structure their configuration via a wizard-based form-filling sequence instead of using one form as shown in figure (3).

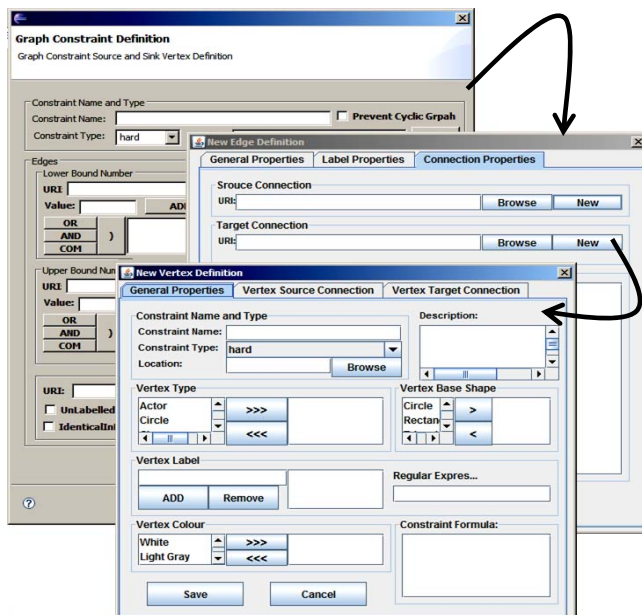


**Figure 3: Using wizard to define a graph constraint that refers to an edge constraint which refers to a vertex constraint.**

To define a constraint using the wizard, the user first decides the type of constraint to be defined, vertex, edge, or graph constraint. The user follows the wizard and assigns values to different properties as required. If the value is another constraint, another form can be filled or the user can browse to reuse a previously define constraint.

### 5.2 Constraint Definition by Example

DECS adopts a synergistic approach between the user and the system to define a constraint by example. The user starts by providing one or more examples that express the required constraint and asks the system to try to infer the intended constraint. However, because any example can hold different meanings, the system usually infers different interpretations for the same example. These interpretations are presented to the user asking for help in specifying the required constraint. If the required constraint is in the inferred list, the user chooses the constraint and confirms by pressing “YES” button.

Figure (4) shows the user trying to define by example the constraint “it is not allowed to connect two vertices of type Actor using an Association edge type”. The user provides a negative example (will be discussed later) showing two Actors connected using an Association edge. The system infers different interpretations for the example including the required constraint which is the first choice in the list. The user selects it and presses “YES” button. Finally, the system generates the XML constraint file to be used at CASE level.

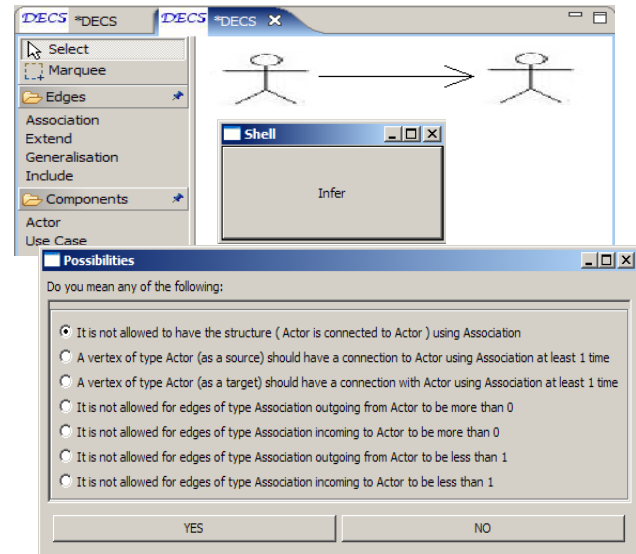


**Figure 4: The user introduces an example and the system shows a list of the inferred interpretations.**

To show some features of DECS, another more complicated example will be discussed. Assume the user wants to define the constraint “it is not allowed to connect Use Case as a source with Actor as a target using any edge type”. In this case the user provides two negative examples using Use Case as a source vertex, Actor as a target vertex and any two of the diagram edge types, Association and Include as an example figure 5a. The system in this case assumes that the user introduced only one example and based on that infers the constraint “both introduced structures must not be in the diagram together”. As the user could not find the required constraint in the list, s/he presses “NO” button. The system understands that the user provided two examples and makes a visual generalisation. Visual generalisation has been used in a different context by [1]. DECS has a novel visual generalisation technique as it shows the user visually the generalisation result figure 5. It remodels the example by fusing the generalised element, in this example, the

edge types figure 5b. It specifies the generalised types by a label in the remodelled example. In this example the system generalises the edge types and specifies them by the label “Association OR Include”.

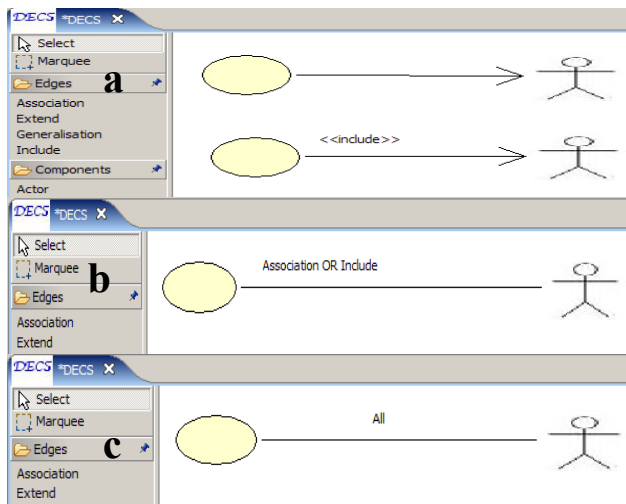


**Figure 5: Visual generalisation. (a) example by the user, (b) first remodelling, (c) second remodelling.**

The system provides the user with a list of constraints again based on the remodelled example. However, the inferred constraint list again would not contain the required constraint. The user presses the “NO” button again. The system understands that the user has provided two edge types as examples and the required is all the available edge types. Based on that, it generalises visually again and converts the label on the edge to be “All” meaning all available edge types figure 5c. The system interprets the remodelled example again and provides the user with a constraint list that contains the intended constraint.

The above example illustrates the synergistic approach adopted by DECS. This synergism between the user and the system keeps the user in the loop of the inference process by helping the system and directing it to the correct inference. A similar synergistic approach has been used by [7] to direct a heuristic search. Keeping the user in the loop is an advantage that has been supported by [7] and [18]. This approach also reduces the number of examples required to express the constraint. As shown above in figure (4), it is possible to express a constraint using only one example because the system expects help from the user. It also reduces the complexity of the inference engine implementation [16].

DECS allows the user to express the required constraint either positively or negatively. A negative example allows the user to show the undesired state, as in the examples above, while a positive example expresses the required state. When the system infers, it interprets the example positively and negatively and presents all the alternatives of both interpretations to the user in the constraint list to choose from.

### 5.2.1 Inference in DECS

DECS has a simple rule based inference engine. Rules are expressed as IF-THEN strings. At runtime, a parser works to convert strings to runtime objects. The inference manager

component is responsible for managing the inference process and encapsulates the inference engine. This component is only effective when the CASE tool designer is in the constraint definition by example environment.

The DECS inference engine has two types of rules called “choice” and “remodelling”. “Choice” rules are responsible for generating the inferred constraint list presented to the user. “Remodelling” rules are responsible for the visual generalisation and example remodelling presented in figure 5. They perform the action of modifying the example by fusing the elements and generating the appropriate labels.

## 6. Experimental Methodology

### 6.1 Aim and Hypothesis

The aim of this research is to reduce the complexity and to improve the process of CASE tool specification by facilitating and simplifying the constraint definition process within the context of meta-CASE tools using constraint specification by example. For this purpose, an experiment has been conducted to evaluate the novel technique, Specification by Example, in comparison to the common, form filling, technique (using a ‘wizard’). The following potential dependant variable measurements were tested for each technique.

- The performance in terms of the resulting constraint specification correctness.
- The time required for accomplishing the constraint specification task.
- The satisfaction of the user with the technique.

The null hypothesis of this experiment is that there is no difference between the techniques regarding the variables to be measured. The independent variables of the experiment are the two techniques, form-filling (wizard) and SBE, while the dependant variables are the correctness and the time required to accomplish the constraint definition task.

### 6.2 Collection and Tasks

The study required users to carry out a set of supplied constraint definition tasks. Two diagram types were selected: State Transition Diagram (hereafter STD) and Use Case Diagram (hereafter UCD). The experiment was conducted on both diagrams separately and at different times. The experiment on STD was conducted in Scotland while the UCD experiment was conducted in Jordan and in Scotland and, consequently, had more participants. Both diagrams were selected because they are commonly used in the design process and all the participants were familiar with them. The diagrams also contain by default all the general constraints that may appear in most other diagrams.

41 participants (16 used STD and 25 used UCD) were selected from Computing Science and Software Engineering graduates and postgraduate students in Scotland and in Jordan. In the Scottish study, one final year undergraduate student also participated. The difference in number of participants for each diagram refers to use extra 9 participants on UCD for more data validation.

A main constraint list was created for each diagram. Each list contained constraints that define a diagram language. Some other constraints were added to each list as customised constraints that define potential customised requirements to take advantage of the power of DECS as a meta-CASE tool. Constraints in each list

were divided into six different groups based on similarities between constraints. The constraint groups were divided based on the following criteria:

- constraints related to cardinality of vertices. This class contains constraints that define the upper bound and lower bound numbers of allowed vertices such as the constraint in figure 2, “*it is not allowed to have more than one Start State in the diagram*”.
- cardinality of incoming and outgoing edges.
- connection between vertices such as the constraint in figure 4, “*it is not allowed to connect two vertices of type Actor using an Association edge type*”.
- unique visual representation
- and label-related constraints.

Some of these criteria generated more than one group such as the first two that contain two groups each: upper bound and lower bound number. One criterion, unique visual representation, forms a group by itself and was used only in STD.

From each group a constraint was selected randomly to form a small list of six constraints. The selected constraint was removed from the group to ensure participation of all the constraints in the lists. When a group has no more constraints, all the removed constraints were returned back to start choosing again. Of course each group has a different assigned number of constraints to select from. Using this method of grouping and constraint selection, it was guaranteed that all the users will get similar, but not identical, lists of constraints. This reduces the differences between the constraint lists with different users and used a wide range of constraints for each diagram type. The generated small lists with 6 constraints each were used in the experiment as tasks; each list containing 6 constraints is a *task*.

### 6.3 Experimental Design

For this evaluation a within subject design was adopted. Constraint lists were randomly assigned to different participants. For each participant two different constraint lists were used, one for training and the other for the constraint definition task. The users’ task was to define the 6 constraints in the list provided. The same list was assigned to each subject for form-filling and for SBE techniques to eliminate any effect of single constraints on the technique evaluation. The order of technique usage was counterbalanced to avoid any order effect.

Each participant was trained using the training constraint list for about 20 minutes on each technique. The training included explaining the technique, performing the task while the participant was watching, and allowing the participant to carry out the training task with all the required assistance and discussion of answers before starting the main task. Participants were asked to fill out a number of different questionnaires at different stages of the experiment. A time limit of 15 minutes was imposed for each task in each technique. The time limit was decided based on preliminary pilot studies which showed that 15 minutes is a reasonable time. Each evaluation was recorded by screen capture to be used later for data extraction and analysis. The experimenter also observed each session and took notes.

It should be noted that the experimenter intervened on several occasions when participants exhibited signs that they could not progress with the tasks. This only occurred in the wizard

condition and help was solely in the form of hints about the labels of different properties in the wizard. No further interference or help was provided.

## 7. Results

All the 41 participants were familiar with the diagrams used in the evaluation. Most of the participants indicated that they were familiar to some degree with the constraint definition task with an average of 3 and 4 on a scale of 5 for STD and UCD respectively. Each experiment took 1.5 hours including the training and questionnaire filling time. The results for the users’ attempts were analysed with respect to the above mentioned hypothesis. No normality assumptions were made about the collected data. Consequently, the nonparametric Mann-Whitney test was used to analyse correctness and satisfaction results while survival analysis with log-rank test was used to analyse the time results. The comparison between the two techniques regarding the required measurements is presented in the following sections.

### 7.1 Correctness

The number of correctly defined constraints for each participant in both diagrams was gathered from the recorded screen capture videos. Constraints that have not been attempted by a participant because of the time limit (the 15 minutes given to accomplish the task) were considered as defined wrongly. No constraints were not attempted for any reason other than the time limit. Figure 6 shows the percentage of correctly defined constraints for each user in STD and Figure 7 shows the same results for the UCD. Both figures show the results for wizard and by example techniques. A significantly higher number of constraints were specified correctly using SBE than via the wizard.

#### 7.1.1 State Transition Diagram

In case of the STD, 12 out of 16 (75%) of the participants defined a higher percentage of constraints correctly using the by example technique than the wizard. Nine users defined all the constraints correctly in the task using by example technique, two of whom also did using the wizard (figure 6).

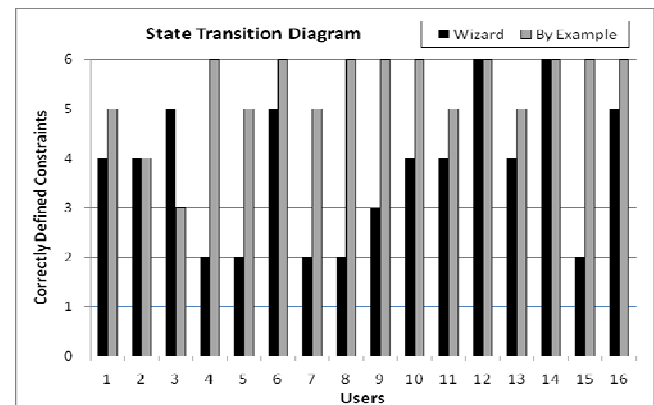


**Figure 6. Number of constraints defined correctly using wizard and SBE by each user in the STD.**

The average percentage of constraints defined correctly using the wizard technique was 62.5% (3.8 constraints) while it was 89.6% (5.4 constraints) using by example. Analysis shows that there is a

high significant difference between both techniques regarding correctness in constraint definition ( $p = 0.001$ ).

### 7.1.2 Use Case Diagram

In case of UCD, 24 users (96%) defined a higher number of constraints correctly using the by example technique than the wizard technique. Only one user defined the same number of constraints correctly using both techniques. 19 users defined all the constraints correctly in the task using the SBE technique, and one only did using the wizard (figure 7).

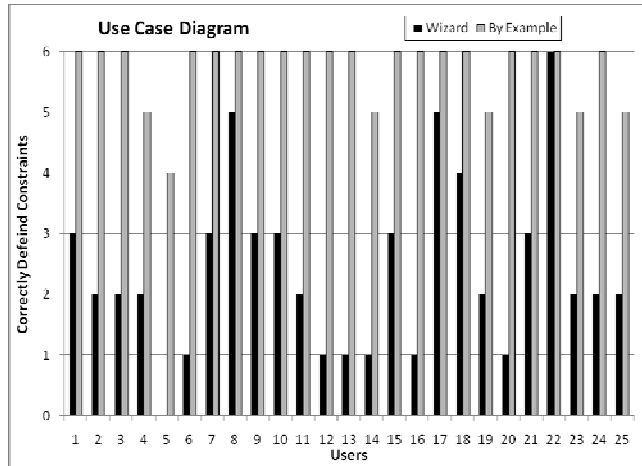


**Figure 7. Number of constraints defined correctly using wizard and SBE by each user in the UCD.**

In the case of the wizard, the average percentage of constraints defined correctly was 40% (2.4 constraints out of 6) while it was 95% (5.7 constraints out of 6) using SBE. Analysis shows that there is a high significant difference between both techniques regarding the correctness in constraint definition ( $p < 0.001$ ).

## 7.2 Time

The time required to accomplish the constraint definition task for each participant in both diagrams was gathered from the recorded screen videos. The time for each participant was rounded by a factor of 15 seconds. This means if a participant accomplished the task using 10 minutes and 14 seconds, that was rounded to 10 minutes but if the time used was 10 minutes and 15 seconds, this was rounded to 10.5 minutes. Figure (8) shows the time required for each user to accomplish the task in the STD and Figure (10) shows the same results for the UCD. Both figures show the time for the wizard and the SBE techniques. In both diagrams it is clear that SBE produced better results than the wizard.

As mentioned above, 15 minutes were given for each participant to accomplish the task. However, some of users did not finish the task within the 15 minutes and some others required all the time and finished exactly after 15 minutes. For these users, a special statistical analysis, survival analysis, was required to measure the significance of difference between both techniques regarding the time required to accomplish the task. If such data have been ignored or excluded from the data analysis, this would introduce a selection bias to the experiment. In survival analysis, the data for participants that did not finish their tasks or subjects that stopped doing the experiment before the measured event has happened to

them is called right-censored data. Survival analysis using Kaplan-Meier survival curves, which show the percentage of the population still surviving at a giving time point, was conducted on the time data for STD and UCD and presented in figures 9 and 11 respectively. Long-rank test was used to calculate the significant difference between the two Kaplan-Meier survival curves for both techniques.

### 7.2.1 State Transition Diagram

In case of the STD, 50.0% (8 of 16) of users accomplished the task in less time using the SBE technique while 37.5% (6 of 16) of users accomplished the task in less time using the wizard. Three users required the 15 minutes maximum time and other three required more than 15 min to accomplish the task in the wizard. Only 1 required the 15 minutes and 3 required more in case of SBE (figure 8).

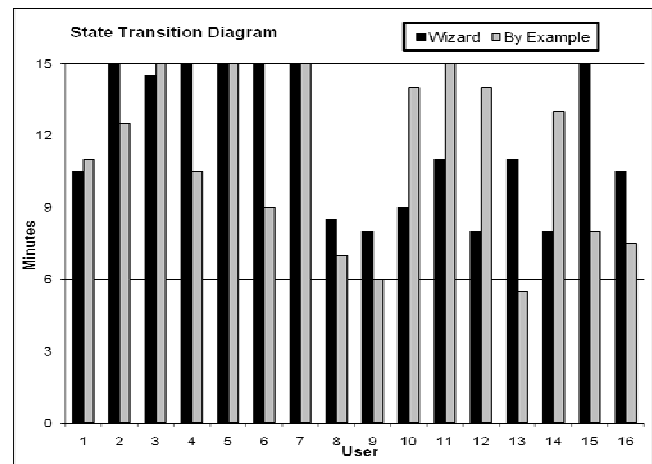


**Figure 8. Task completion time in minutes for Wizard and SBE by each user in the STD.**

Two users required the 15 minutes allowed or more in both techniques and they were the only users who required the same time for both techniques. One of them finished the task using the maximum 15 minutes in SBE but not in wizard while the other did not finish the task in either case.

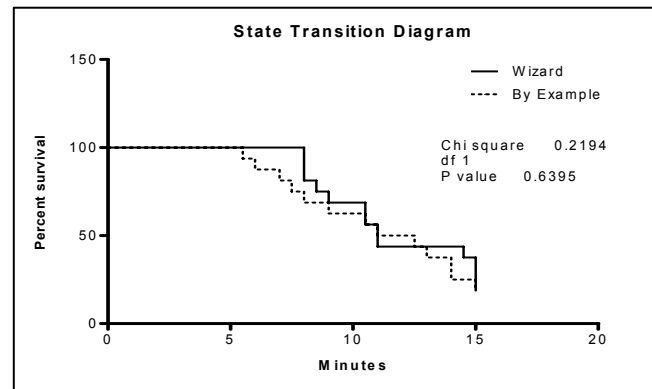


**Figure 9. Kaplan-Meier survival curve of STD.**

The Kaplan-Meier survival curve for STD (figure 9) shows that both curves are close to each other and have almost the same trend. However, it also shows clearly that SBE curve is under the

wizard almost all the time which indicates that higher number of users finished their tasks using SBE in less time as also figure 8 shows. P value (= 0.64) shows that there is no significant difference between both techniques regarding the required time to accomplish the task.

### 7.2.2 Use Case Diagram

In case of the UCD, 88% (22 of 25) of users accomplished the task in less time using SBE while two users required the same time, 14 and 10 minutes, in both techniques.

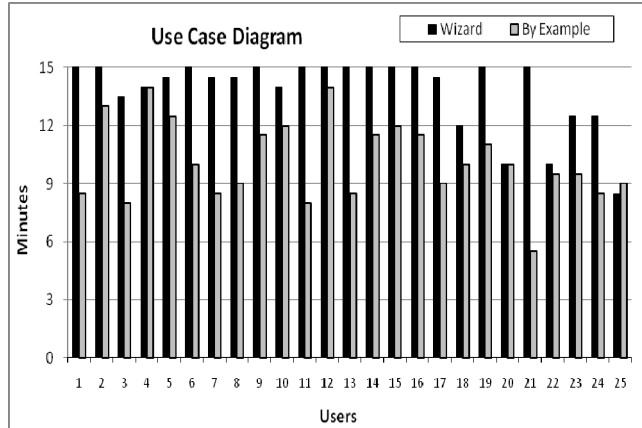


**Figure 10. Task completion time in minutes for the Wizard and SBE by each user in the UCD.**

11 users used more than the 15 minutes given and one required exactly 15 minutes using the wizard technique; while all the participants finished before consuming the given time in the case of SBE. The results suggest that participants have performed more quickly with the specification by example technique than the wizard (figure 10).

The Kaplan-Meier survival curve for UCD, (figure 11), shows a distinct difference between both technique curves. SBE survival curve is always under the wizard one which indicates that number of participants accomplished the task using SBE is higher.

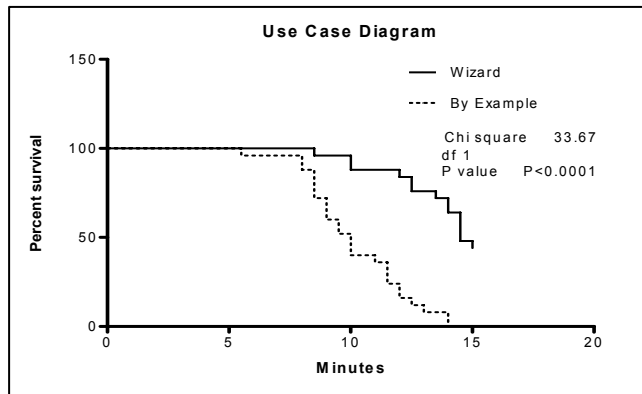


**Figure 11. Kaplan-Meier survival curve of UCD.**

The figure also shows that none of the participants has failed to accomplish the task within the required time using SBE technique. P value (<0.0001) shows that there is a high

significant difference between both techniques in time required to accomplish the task.

### 7.3 User Perception

In order to provide further validation for the hypothesis, post task and post experiment questionnaires that participants filled out were analysed.

**Table 1. User perception of wizard and SBE techniques in STD and UCD. (higher = better)**

| Diagram                                                                                 | STD |     | UCD |     |
|-----------------------------------------------------------------------------------------|-----|-----|-----|-----|
| Questions                                                                               | Wz  | SBE | Wz  | SBE |
| How successful were you in accomplishing what you were asked to do?                     | 3.7 | 4.3 | 2.5 | 4.8 |
| This technique was powerful enough to allow me to define my constraints.                | 3.9 | 4.3 | 4.3 | 4.7 |
| In most cases, I was confident that I defined the required constraint.                  | 3.3 | 4.6 | 2.4 | 4.8 |
| I felt that I acquired experience in this technique quickly while I was working.        | 4   | 4   | 3.6 | 5   |
| I am satisfied with my performance in constraint definition tasks using this technique. | 3.5 | 4.1 | 2.4 | 4.8 |

**Table 2. User perception of wizard and SBE techniques in STD and UCD. (lower = better)**

| Diagram                                                                | STD |     | UCD |     |
|------------------------------------------------------------------------|-----|-----|-----|-----|
| Questions                                                              | Wz  | SBE | Wz  | SBE |
| How mentally demanding was the task using this technique?              | 3.5 | 3.1 | 4.2 | 1.4 |
| How hurried or rushed was the pace of the task using this technique?   | 2.8 | 2.3 | 3   | 1.2 |
| How hard did you have to work to accomplish your level of performance? | 3.1 | 2.4 | 3.9 | 1.4 |
| How uncertain, discouraged, irritated, stressed, and annoyed were you? | 2.6 | 1.8 | 3.8 | 1.2 |
| While I was working, I felt that I needed help from an expert.         | 2.9 | 2.1 | 4.3 | 1.3 |

In post task questionnaires (filled in after each technique), participants' opinions about the constraint definition task using each technique were investigated. There was a need to discover the feelings of the participants while interacting with the system in case of each technique. A Likert 5-point scale was used and some of these were inverted to reduce bias. Tables (1 and 2) above show some of the questions and the average answer

numbers on the scale. Table 1 shows questions where the higher scale answer is better while Table 2 shows questions where the lower scale answer is better in both diagrams.

## 8. Discussion

A higher percentage of constraints have been defined correctly using SBE than using a wizard. A statistically significant difference appeared between the two techniques in both diagrams. The explanation of this result can be referred to the ability of the users to express constraints visually more easily than expressing them through filling a form of properties, especially when URI references are used. Anecdotal evidence supporting this idea is the observation that the users, while using the wizard, drew example constraints, especially connection constraints, using pen and paper before starting the definition process. As mentioned above, selecting the order of technique usage was random. So if the user used the wizard technique first, s/he started using constraint drawing even before being introduced to the SBE technique. In the case that a user had already used SBE before the wizard, this can be explained as the effect of the technique on them. This gives an indication that visualising the constraint provides better understanding for its concept, which supports using SBE and explains its effectiveness in the constraint definition task.

In general, participants required less time to define constraints using SBE than the wizard. This result is easily explained by mentioning that the wizard, as any form-filling technique, needs more time in reading the property labels and exploring the alternative values. This time is reduced in the case of SBE. On the other hand, the SBE technique consumes the user's time in thinking of suitable example(s) to express the constraint. However, SBE outperformed the wizard in time measurement. It can be argued that with experience, properties in the wizard can be memorised while the time for expressing a constraint using SBE will stay constant which may overturn the experimental result. The answer here is that observations and qualitative interviews showed that the user gets experience in SBE quicker than the wizard (Table 1). Although it is not possible to generalise these results over long system usage and a large number of constraint definition tasks, they suggest that users can get experience in SBE even if it depends on introducing different examples for each constraint.

As shown in the survival analysis, there was no significant difference between the two techniques regarding the time required to accomplish the task in the case of STD. By contrast, a strong significant difference appeared in the case of UCD for the same measurement. It is difficult to explain this data. As mentioned above, the significant difference can be justified as the SBE technique is easier than wizard and it is possible to accomplish the task in less time. This is clear from the curves in both diagrams. Constraint lists provided as task cannot clearly justify the difference between the diagrams because the constraints were of the same types. However, the individual constraints were, of course, different and may have different levels of difficulty. This difference in results between the two diagrams requires more investigation.

Regarding user satisfaction, it is clear that users prefer the SBE technique and were more satisfied using it compared to the wizard. Tables 1 and 2 are self explanatory; users felt less

stressed, less rushed, less uncertain, and more confident while performing tasks using SBE. This might explain the better performance in correctness and time especially as they believed that they have done better in SBE in a question not included in the tables. It is important to note here that users agreed that both techniques were powerful enough to define constraints with insignificant advantage to SBE ( $p = 0.626$  and  $p = 0.255$  for STD and UCD respectively). Some questions in the post experiment questionnaire (filled in after both techniques) asked which technique was easier to be learned and remembered, which technique will you use for constraint definition if you were assigned such task, and which technique will be more effective? All their answers were to the advantage of SBE.

The above results demonstrate that using SBE for constraint definition in the context of meta-CASE tool reduces the complexity of the constraint definition task. This also indicates that the SBE technique can add value to the meta-modelling process and consequently to the CASE specification process in general. Consequently, the study rejects the null hypothesis.

An interesting observation is that during the experiment, users understood the inference engine of the system and expected the choices that appeared in the list. Because the same constraint suggestions can appear for different examples, participants came to expect the suggestion would appear in the same place. For example, constraints that deal with labels always appear at the bottom of the list. So users were starting search the list from the bottom instead of starting from the top based on their experience during the training time. This suggests that it may be problematic to allow an inference engine to rank (and hence list) the suggestions based only on their likelihood in a particular context.

### 8.1 Threats to Validity

Limiting the task time to 15 minutes could have an effect on the correctness measurement. If subjects were permitted to continue without time limitation, some of them "possibly" would have got higher correctness. However, extending the time limit would not threaten the validity of the significant result for UCD as all subjects completed the task within 15 minutes using SBE. It seems highly unlikely that extending the time limit for STD would produce a significant result but this needs further investigation. Intervention (mentioned in section 6.3) also threatens the validity of the results. However, it only occurred in the wizard condition which worked against the SBE technique. Since the SBE technique outperformed the wizard, this indicates that the intervention had very little or no effect. Subjects' native language is also a threat to validity as not all the subjects are native English speakers. However, all the participants had or were carrying out their studies in English. These are examples of the internal threats to this research.

With respect to external threats, using only two diagram types (UCD and STD) may limit the generality of the results. The decision to use two diagram types was taken to reduce the threat of bias compared to using only one type. Additionally, time and resource limitations prevented investigation of more diagram types.

## 9. Related Work

Several constraint classifications have been introduced by [6] and many of these classifications have been adopted into DECS, such

as hard and soft constraints [19]. Their paper also introduced the idea of a two level specification model depending on generating a constraint based tool specified using a meta-CASE tool and can be reconfigured and customised by the generated tool user through constraint manipulation. Scott, Horvath, & Day [20] introduces the notion of using property-value but in different context and classification adopted in DECS. Their classification is general and complex compared to that in DECS and they have a wide range of values for each property which can be considered as complication. Offen [17] and [20] introduce a CASE tool, CASEMaker, that customise constraints but depend on a graphical meta-model and uses a constraint language to define constraints.

Some empirical studies were conducted in the context of meta-CASE tool and constraints. One experiment [17] was to evaluate the effect of constraints on the work of designers in CASE tools. A meta-CASE tool was built, "CASEMaker", to generate constraint-dependent CASE tools and conduct the study. Another experiment was conducted by [13] to test a PBE application in a drawing tool, Metamouse. This tool is distinguished regarding its ability to learn from examples the user introduces which is called "learning by example". However, Metamouse, learns only repetitive tasks by inferring and generalising constraints. Through an empirical study with six tasks assigned to participants, [13] intended to know if users can teach Metamouse by example or not. In the study, they measured time to accomplish the task and get quantitative and qualitative data. They claim that the last was more helpful to them. Results show that Metamouse needs enhancement.

An interesting study is done by [15] for evaluating the PBE system Peridot. The aim of the study was to evaluate how easy Peridot is to use. Ten people, half of them are programmers and the others are not, participated in the experiment. Time for building a menu task was calculated and results showed that Peridot is efficient as it reduces the time required to build a menu from 50-500 minutes to 4-15 minutes.

One of the attractive meta-CASE tools is Pounamu [22]. The tool depends on class diagram as a meta-model and a separate event handler. The event handler is used to define constraints using a graphical user interface with checkboxes to specify the event. However, the response to the event is introduced by the user through Java code. The same tool has been supplied with a visual event handler tool [12]. The visual event handler uses a visual language to model events and responses. The model uses visual constructs and blocks which form at the end a model similar to the flow diagram modelling an event and response. One interesting point in this visual tool is that it encapsulates a sequence of actions (as a response) into one visual block and they call that "packing". They say "packing complex parts of a specification into reusable building blocks allows very complex event handlers to be defined with the model". It is a similar idea to the nesting of constraints at DECS which also allows the specification of complicated constraints.

KidSim is another interesting tool. It is not a meta-CASE tool; however, it is reconfigurable using two techniques, graphical rewrite rules and programming by demonstration [21]. One important aspect here is that they address the problem of generalisation and they handle that using a pull-down menu from which the user chooses the required generalisation.

The idea of depending on a constraint manager to be able to introduce two specification levels using constraints has been presented by [6] but has not been implemented by them before. MetaBuilder [5] and CASEMaker [18, 15] are two similar tools that use a technique for designer activities monitoring like constraint manager at DECS. They use hypernode model and constraints for methodology modelling. Gong, Scott, & Offen [5] explicitly specifies the use of a constraint type checking technique. This technique works to ensure that design work does not violate the defined methodology. The diagramming environment, type checker, and the methodology definition store are three separate components. This allows the ability to modify the methodology in the repository and the modification will be reflected on the diagramming environment without need for re-compiling or re-generating it. This is because the type checker will keep checking the diagram according to what is defined in the methodology definition store (repository). In both tools, they provide a user interface to define constraints over a hypernode. In a similar way, constraint manager in DECS builds the constraints from XML files only once at the first time the environment is used. After that it maintains the constraints in the memory as runtime objects. It is believed that this technique is more efficient, regarding time required, than checking XML files each time looking for a violation.

## 10. Conclusions and Future Work

This paper has presented an empirical study comparing two constraint definition techniques, a conventional form-filling technique represented by a wizard and Specification by Example (SBE) technique, the application of programming by example (PBE) technique to model specification. The latter has not been used in the context of meta-CASE tools before. The study evaluated both techniques using two diagrams, a State Transition Diagram and a Use Case Diagram, with respect to constraint definition correctness, required time to accomplish a constraint definition task, and user satisfaction. Results show that in general, the SBE technique outperforms the wizard in all the measured criteria.

The DECS version described here and on which the experiment was conducted allows the user to provide an example and it interprets it positively and negatively showing the inference results in one list. Originally the system was designed to infer negative examples; however, it was found that implementing positive example interpretation will make the system more flexible. Allowing positive and negative examples was appreciated by the user and many of them described the system as intuitive. Some other users described that as confusing. In a later version of DECS (implemented after this experiment finished), the user was given the choice to ask the system for positive or negative interpretation of the introduced example. This filtration, theoretically, has an advantage of reducing the number of presented choices the user should search in.

The new DECS version also allows the user to switch on and off the inference engine at anytime during the example construction in addition to switching between positive and negative example interpretations. This transparency increases the user understanding of the system inference process.

An empirical study is under design and will be conducted to study the benefit of these two implemented features. A preliminary

study (as online questionnaire) has been launched to evaluate the complexity of understanding the constraints expressed positively and negatively in English, which is different from the positive and negative examples discussed above. However, part of the intended experiment will study the relation between the expression of the constraint in English and the example that the user thinks in and provided to the system to express the constraint.

As mentioned above, CASE tool specification is a potentially large and ill-defined decision space. A similar problem has been discussed in [11]. Their solution depended on limiting the number of constraints that can be inferred. Since DECS generates constraint-based CASE tools, it is not possible to adopt a similar solution as that may limit the number of constraints or their expressiveness. It is also impossible to predict all the possible constraints from the same example as this may result of a high number of alternatives. To solve this issue, trials have started to implement a technique to teach the system how to infer required constraints. From another perspective, this augments the inference engine with extra rules without the need to write code. This issue will also be evaluated using an empirical study.

Finally, only two diagram types have been examined in this study. Future work will look at other diagram types and consider the potential affect of the diagram type or the context of use on the usefulness of the constraint definition by example approach.

## 11. REFERENCES

- [1] Amant, R., Lieberman, H., Potter, R., & Zettlemoyer, L. (2000). Visual Generalisation in Programming By Example. *Communications of the ACM* , 43 (3) , 107 - 114.
- [2] Argall, B., Chernova, S., Veloso, M., & Browning, B. (2009). A survey of robot learning from demonstration. *Robotics and Autonomous Systems* , 57 (5), 469-483.
- [3] Ferguson, R., & Hunter, A. (2000). *MetaBuilder: The Diagrammer's Diagrammer*. Lecture Notes In Computer Science; Vol. 1889 (pp. 407 - 421). London, UK: Springer-Verlag.
- [4] Findeisen, P. (1994). *The Metaview System*. University of Alberta. Alberta: University of Alberta.
- [5] Gong, M., Scott, L., & Offen, R. (1997). *MetaBuilder: a generic CASE tool builder*. Software Engineering Conference, 1997. Asia Pacific and International Computer Science Conference 1997. APSEC '97 and ICSC '97. Proceedings, (pp. 435-444).
- [6] Gray, P., & Welland, R. (1999). Increasing the flexibility of modelling tools via constraint-based specification. *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. Mississauga, Ontario, Canada: IBM Press.
- [7] Hudson, S., & Hsi, C.-N. (1993). A Synergistic Approach to Specifying Simple Number Independent Layouts by Example. *Conference on Human Factors in Computing Systems* (pp. 285-292). New York, USA: ACM.
- [8] Isazadeh, H., & Lamb, D. A. (1997). *CASE Environments and MetaCASE Tools*. Kingston, Ontario, Canada: Department of Computing and Information Science, Queen's University.
- [9] Jankowski, D. (1997). Computer-Aided Systems Engineering Methodology Support and Its Effect on the Output of Structured Analysis. *Empirical Softw. Eng.* , 2 (1), 11-38.
- [10] Kelly, S. (2009). *Domain-Specific Modelling: A Toolmaker Perspective*. Retrieved March 4, 2009, from MetaEdit+ Workbench questions: <http://www.metacase.com/faq/showfaq.asp?cate=MWB>
- [11] Kurlander, D., & Feiner, S. (1993). Inferring Constraints From Multiple Snapshots. *ACM Transactions on Graphics* , 12 (4), 277-304.
- [12] Liu, N., Hosking, J., & Grundy, J. (2007). A Visual Language and Environment for Specifying User Interface Event Handling in Design Tools. *Conferences in Research and Practice in Information Technology Series*; Vol. 241 (pp. 87 - 94). Ballarat, Victoria, Australia : Australian Computer Society, Inc., Australia.
- [13] Maulsby, D., & Witten, I. (1993). *Metamouse: An Instructivle Agent for Programming by Demonstration*. In A. Cypher, Watch What I Do: Programming by Demonstration. Cambridge: MIT Press.
- [14] MetaCase. (2009). *MetaEdit+*. Retrieved 7 5, 2009, from <http://www.metacase.com/>
- [15] Myers, B. (1993). *Peridot: Creating User Interfaces by Demonstration*. In A. Cypher, Watch What I Do: Programming by Demonstration. Cambridge, Mass.: MIT Press.
- [16] Myers, B., McDaniel, R., & Wolber, D. (2000). Intelligence in Demonstrational. *Communications of the ACM* , 43 (3), 82-89.
- [17] Offen, R. (2000). *CASE Tools and Constraints*. North Ryde: Macquarie University Joint Research Centre for Advanced Systems Engineering.
- [18] Prabaker, M., Bergman, L., & Castelli, V. (2006). An Evaluation of Using Programming by Demonstration and Guided Walkthrough Techniques for Authoring and Utilizing Documentation. *Conference on Human Factors in Computing Systems, Proceedings of the SIGCHI conference on Human Factors in computing systems* (pp. 241 - 250). Montréal, Québec, Canada: ACM, New York, USA.
- [19] Qattous, H. (2009). Constraint Specification by Example in a Meta-CASE Tool. *Foundations of Software Engineering, Proceedings of the doctoral symposium for ESEC/FSE on Doctoral Symposium* (pp. 13-16). Amsterdam: ACM.
- [20] Scott, L., Horvath, L., & Day, D. (2000). Characterising CASE Constraints. *Communications of the ACM* , 43 (11), 232-238.
- [21] Smith, D., Cypher, A., & Spohrer, J. (1994). *KidSim: programming agents without a programming language*. *Communications of the ACM* , 37, 54-67.
- [22] Zhu, N., Grundy, J., & Hosking, J. (2004). Pounamu: A Meta-Tool For Multi-View Visual Language Environment Construction. *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing* (pp. 254-256). IEEE Computer Society.