ViKER: A Visual Interface for Transformations Between EER and AR Conceptual Models

|  |  |
| --- | --- |
| St John Grimbly  UCT  Department of Computer Science  GRMSTJ001@myuct.ac.za | Jeremy du Plessis  UCT  Department of Computer Science  DPLJER001@myuct.ac.za |
| Gabriel Stein  UCT  Department of Computer Science  STNGAB004@myuct.ac.za | |

This paper investigated the… We found that…This is useful because…

# Introduction

Traditional data management modelling procedures involves designing a sequence of requirements in the form of a conceptual model such as an entity relational model (ERM). This conceptual model is then transformed into a relational model (RM). The RM is more representative of the actual implementation of the database. This database is then created and put into production, whereby the design models are then discarded.

Interesting ideas for reusing the valuable information presented in the conceptual models have arisen in the past, including using conceptual models to ‘query-by-design’ using UML-like notation. This would make understanding and interfacing with the underlying database much easier for non-expert users.

The open-world assumption is that the assumption that a statement cannot be known to be false unless it is explicitly stated as such. It is the exact opposite of the closed-world assumption. The open world assumption is traditionally …

This project investigates methods of allowing non-expert database users the ability to easily transform between abstract relational models and extended entity relational models. More specifically, this project investigates methods of transforming between EER and ARM by implementing the theoretical transformation rules outlined in the KnowID paper (incl. references). Focus is placed on implementing clean code for future extension by other teams, properly documenting code and implemting a user friendly interface for easily performing transformations implemented in this paper.

# Theory

The KnowID (insert reference) paper introduces a new transformation procedure for converting between abstract relational models (ARM) and enhanced entity relational models (EER) models. ARM is an extension of RM presented in reference.

ARM extends traditional RM diagrams by including abstract datatypes - OID - which act as memory references to other relations in the model. Each relation is assigned a ‘self’ reference which uniquely identifies it. Attributes of the relation are thus not primary keys and we can be assured that every relation has a primary key – self. Attributes can be identifiers, forming part of a path functional dependency. This pfd->self.

EER extends traditional ER diagrams by including notation for specialisation, partitioning, generalisation and aggregation relationships. It is thus a more representative conceptual model for the underlying database structure and makes ideas such as ‘query-by-design’ easier to implement (reference).

This project makes use of JavaScript

# Requirements Captured

The scope of this project was open-ended in that the client did not specify an exact end product that was expected to be produced. The client did, however, have some key requirements for the end product. These were:

**Hard requirements**

1. Implement the rules as in the KnowID paper, both the EER to ARM and ARM to EER.
2. Report on success/failure of a transformation.
3. A user interface.
4. Open/save models.

**Soft requirements**

1. Report on those things that could not be transformed.
2. Report on what happened with each element.
3. Textual or graphical representation of the models.

**Engineering**

1. Permissible to extend a current open source EER tool.
2. Textual representation (perhaps) with XML

Insert sequence diagrams showing these requirements captured

**User Case Narratives**

1. Convert ER model to AR model

The user (Primary Actor) will select elements from the Entity Relationship elements tab (where the elements are displayed to be dragged onto the action area) and drag-n-drop their selected elements on to the action area and create an entity relationship (ER) model of their choosing by linking the elements. They will then click the ‘Transform’ button and the system will render an abstract relationship (AR) model. The one alternative flow to a direct transformation, is if the user created an ER model that cannot be transformed into an AR model due to not fitting the transformation rules. In that case, the error will be reported in the error log below the action area. Another alternative flow is if the user created an ER model that can only be partially transformed into an AR model in which case whatever transformation cannot be performed, will be outputted in the error log below.

1. Convert AR model to ER model

The user (Primary Actor) will select elements from the Abstract Relationship elements tab (where the elements are displayed to be dragged onto the action area) and drag-n-drop their selected elements on to the action area and create an abstract relationship (AR) model of their choosing by linking the elements. They will then click the ‘Transform’ button and the system will render an entity relationship (ER) model. The one alternative flow to a direct transformation, is if the user created an AR model that cannot be transformed into an ER model due to not fitting the transformation rules. In that case, the error will be reported in the error log below the action area. Another alternative flow is if the user created an AR model that can only be partially transformed into an ER model in which case whatever transformation cannot be performed, will be outputted in the error log below.

1. Save model

Once the user (Primary Actor) performs the relevant transformation, they click the ‘Save Model’ button and the system will ask for a file name to save the model in a relevant format on the server. This allows the user to load the model when they come back onto the system at a later stage. The only alternative flow is if the transformation cannot be performed, as the system will not allow the user to save a model that cannot be rendered.

1. Load model

If the user (Primary Actor) wants to load a previously transformed model, they click the ‘Load Model’ button and select a model from the file server that they want to load. The alternative flow to loading a model is creating one from scratch as detailed in the first use case.

1. View error log

When the user (Primary Actor) performs a transformation, the success or failure of the transformation will be outputted in the error log that is located at the bottom of the interface. If the transformation is completed successfully and all transformation can be performed, the log will output ‘Transformed Successfully’. If only some of the transformations can be performed, the system will output which transformation cannot be performed and why not. If none of the transformations can be performed, the log will output all the transformations that cannot be performed and why not. More specific error reporting features will be discussed with the client over the course of development.

USE CASE DIAGRAM

A close up of a logo

Description automatically generated

**Back-end Requirement Captured**

List how each of these were captured.

Special focus was placed on capturing the key requirements of the project. Careful reading of the theoretical discussions of both ARM and EER was done before any design was started. This led to a smooth development process in which all the core requirements set by the client were captured. The captured requirements of the backend are as in the following form:

1. The rules of the transformations have been captured in the EERToARM and ARMToEER classes in which OOP representations of the respective models are transformed. The OOP representations are then transformed into JSON representations in the ReadWriteARM and ReadWriteEER classes.
2. The status of the transformations (success/failure) and information about the transformations are captured during the transformation process in the EERToARM and ARMToEER classes. This is captured as a python dictionary and appended to the JSON representation of the models in the WebServer.
3. The user interface requirement is discussed in detail in the next section. This project created an intuitive UI far beyond the scope of the hard requirements.
4. The open/save functionality was implemented on the front-end. See next section.

**Front-end Requirements Captured**

The front-end was, though not a hard requirement of the client, a key focus in our design process. The client specified that a major reason for this transformation procedure was to allow conceptual models of the database to be useful beyond the design phase. It was thus decided that a user interface would be the most user-friendly experience for an end user, as having to write text based, complex models would be difficult and cumbersome. The features of the project captured on the front-end are as follows:

1. A user can load a JSON model by clicking the ‘Load Model’ button on the web application GUI. This logic is performed in the app class that creates an ERModel or ARModel object to parse the JSON and manipulate it in order to render a graph.
2. The model graph is then rendered in the input section via the EntityGraphModel class.
3. The model can then be transformed from to EER/ARM by clicking the ‘Transform Model.’ Which makes a POST request to our python webserver and gets the transformed model back as JSON.
4. The success or failure of the transformation is reported in a dedicated error log section of the GUI which can later be downloaded by clicking the ‘Save Transformation Report’ button.
5. Detailed information about information lost in the transformation procedure is displayed in the error log.
6. The newly created model can be saved to a JSON representation by clicking the ‘Save Transformation Report’ button.

A screenshot of a cell phone

Description automatically generated

1. Where the input model is rendered
2. Where the output model is rendered
3. Where the error log is printed
4. Button to load model as a JSON file
5. Button to send input model to server and transform to output model
6. Saves the output model as a JSON file along with the error log

The next section deals with the analysis of your system. Cover the functional, non-functional and usability requirements. This is where you present your use case narratives and diagrams.

Discuss the major analysis artefacts that you produced. We will expect you to produce at least one overall description of the architecture used in your system as a diagram, either here or below (see Section 2.3). You may also want to include an analysis class hierarchy diagram.

# Design Overview

**Data Flow**

A screenshot of a social media post

Description automatically generated

A close up of a piece of paper

Description automatically generated**Back-end**

The architecture of the back-end was a key focus in the development of the entire project. Ensuring major changes in design during development were not needed, specific design days were set aside within the first 3 phases of the development cycle.

It was decided early on the use Python as the back-end object-oriented language. Python code is easy to read and implement. There are also many powerful packages available for python such as numpy and json. JSON was chosen as the medium for representing both ARM and EER models in textual form. JSON is easy to use and read – especially when comparing to XML. JSON also is native to JavaScript and thus made it an obvious choice for the front-end.

INSERT JSON FORMATS IN LATEX

CamelCase code convention was used in Python for the sake of consistency across to the front-end.

**Front-end**

The next section is an overview of your design. The system design has to be justified in terms of the expected behaviour of the final product.

If you produced a design class diagram put it here.

You must present the overall architecture of the system together with an architecture diagram. You may choose what kind of diagram best suits your project, but we would expect a layered architecture diagram (see Figure 1) unless there is a good reason for some other kind of diagram. It need not be a formal UML diagram as long as it conveys all the necessary information clearly.

You should then (in subsections) cover the algorithms and the data organisation used and why they were considered the best.

# Implementation

**Back-end**

**Architecture Overview**

The key vision in the design of the system architecture as a whole was to ensure separation of concerns. That is, logic is separated from communication and storage of data. We designed our classes such that the principles of low coupling and high cohesion were followed. To elaborate, we ensured that the elements within a specific class ‘belong’ together in that they are closely correlated in their function or dependency. Further, we ensured that classes themselves are not dependent on each other. They are independent entities in their design.

General architecture buzzword stuff – layered architecture etc.

The UML class diagram in figure shows the class structure of both the front-end and back-end. Inheritance was a key in the choice of structure as both EER and ARM naturally share many fundamental concepts such as attributes.

Classes were designed for attributes, relationships and tables. Specific attributes – ER and ARM – derive from the attribute class. ER entities and ARM relations derive from the table class.

The Main class was responsible for executing transformation logic, calling methods from classes and creating objects as needed. Python Lists, numpy arrays and python dictionaries were the main data structures used throughout the back-end transformation process.

**Classes**

# Main

The main class is responsible for coordinating the transformation logic by calling the appropriate transformation procedures and storing the entities/relations and error log.

# EERToARM

The EERToARM class is responsible for performing the actual transformation from EER to ARM by manipulating the OOP representation of the EER model.

# ARMToEER

As in the EERToARM case, the ARMToEER class is responsible for the transformation of ARM OOP model to EER OOP model.

# Constants

The Constants class is the class in which the enumerated types are defined for our OOP representations of ARM and EER.

# Table

The Table class is the parent class of Relation and Entity. These form the fundamental structures of both ARM and EER models respectively. Relation and Entity inherit from parent as they share concepts of attributes and specific attribute properties as well as Table names.

Relations have…

Entities have…

# Attribute

The Attribute class is the parent class of the ERAttribute and ARMAttribute. As mentioned in the discussion about the Table class, EER and ARM models share the concept of attributes. The attributes themselves do differ in their properties.

ERAttributes have…

Entities have…

# Relationship

The Relationship class is responsible for storing all the appropriate information about a relationship between entities in an EER diagram. Relationships between Relations in ARM can be dynamically determined based on the self\* and foreign keys specified in the relation. In EER, however, one cannot dynamically determine the nature of the relationship between entities. For example, an ExactlyOne to OneToMany relationship provides specific information about the nature of the relationship between the local and foreign entities.

# Test Package

The Test package is responsible for collating all the unit testing and general testing, as well as helper methods required for testing.

ReadWriteJSON

TestEERToARM

TestARMToEER

# WebServer

The WebServer class is the class that allows communication between the front-end visual interface and the back-end transformation logic and associated error reporting. The WebServer is implemented using the Python Flask package. It also makes use of the Python JSON package for manipulating JSON as needed.

The WebServer creates a server endpoint on ***0.0.0.0:5000/api/transform***. This provides both POST and GET procedures. The front-end can send a JSON file to that end-point, at which point the WebServer will determine the JSON type (ARM or EER) and invoke the appropriate transformation procedure. The error log is then fetched and appended to the JSON for front-end logging requirements.

**Front-end**

**Classes**

INSERT SCREENSHOTS

# Index

The index class is responsible for making this a single page application, this means that all components are rendered in one parent component (the index).

# App

The app class contains all subclasses and subcomponents so that the only component being rendered by the index class is the app class. Majority of application state and logic is performed in the app class.

# ERModel

This class is used to parse the JSON data into the EntityModel and EntityAttribute classes so that they can be used in the EntityGraphModel class to render the graph.

# ARModel

This class is used to parse the JSON data into the ARModel and RelationAttribute classes so that they can be used in the RelationGraphModel class to render the graph.

# EntityModel

This class is used to parse the JSON into a useable object that can be rendered in the EntityGraphModel class.

# RelationModel

This class is used to parse the JSON into a useable object that can be rendered in the RelationGraphModel class.

# EntityAttribute

This class is used to parse the JSON into a useable object that can be used to help render the create links between entities and their attributes.

# RelationAttribute

This class is used to parse the JSON into a useable object that can be used to help render the create links between relations.

# EntityGraphModel

This is where the rendering logic is performed in order to render the JSON as a EER as a SVG image.

# RelationGraphModel

A picture containing screenshot

Description automatically generatedThis is where the rendering logic is performed in order to render the JSON as a ARM as a SVG image.

* Describe your data structures and be sure to illustrate them with a diagram.
* If your user interface was a key feature describe how that was implemented.

**Figure 1:** An architecture diagram. Caption to go below figure.   
Note that LibreOffice handles this better than MS word

* Discuss the function of the most significant methods in each class. This may well require flowcharts, or sequence diagrams, in some cases.
* Any special relationship between the classes (e.g. friends) and why they exist.
* A description of any special programming techniques or libraries used.

# Program Validation and Verification

Validation and verification were taken seriously throughout the development plan. During the design phase of the software, we decided to take a test-driven development approach. That is, we would create our critical and functional tests for the software and test our progress against these test cases throughout the development cycles. Test cases were added as needed when a unplanned or unintended task needed to be tested.

Our software consists of two major branches – the front-end and the back-end. Different approaches were taken in the testing of these. User friendliness and user interaction was the top priority in the design of the the interface – discussed in the section. Accuracy of transformation and correct error reporting was the priority for the back-end.

Tell us how you tested the system and why you believe it works. Describe the Quality Management Plan for your project, that is, software testing plan. The plan should indicate the types of testing that was performed and detail how they were done. This must include the reasons on why the chosen testing protocol was considered effective.

Table 1: Summary Testing Plan.

|  |  |
| --- | --- |
| Process | Technique |
| 1. Class Testing: test methods and state behaviour of classes | Random, Partition and White-Box Tests |
| 1. Front-end Integration Testing: test the interaction of sets of classes for the front-end independently of the back-end. | Random and Behavioural Testing |
| 1. Back-end Integration Testing: test the interaction of sets of classes for the back-end independently of the front-end. | Random and Behavioural Testing |
| 1. Validation Testing: test whether customer requirements are satisfied using specific test cases approved by the client | Use-case based black box and Acceptance tests |
| 1. System Testing: test the behaviour of the system as part of a larger environment. Back-end and front-end integration was thoroughly tested. | Recovery, security, stress and performance tests |
| 1. UI/UX Testing: test the user friendliness and intuitiveness of the UI and UX by allowing non-expert users to test the system with minimal guidance. | Anonymous user behavioural testing |

Describe all the steps taken to validate the correctness of the program.

If you had user tests then say what you did and what the results were. Describe why these test data were chosen (what test conditions the data was testing). Table 2 provides an example of the sorts of results we are looking for. The full detail of the test runs should be appended to the report.

Table 2: Summary of tests carried out. A table caption goes above the table.

|  |  |  |  |
| --- | --- | --- | --- |
| Data Set and reason for its choice | Test Cases | | |
| Normal Functioning | Extreme boundary cases | Invalid Data (program should not crash) |
| Preliminary test (see Appendix 3) | Passed | n/a | Fell over |
| Development Unit Testing |  |  |  |
| Final Testing |  |  |  |
| UX/UI User Testing |  |  |  |

Follow your table of results with a discussions of them highlighting how useful and usable your system is for its intended purpose.

# Conclusions

Your report must have a clear conclusion where you revisit the aims set out in the beginning and discuss how well you met them. Did you achieve the objective of creating a well-structured, modular, and robust system? Please summarize the design features and test results that show this.

# Appendix A — Code Legibility and Output

Include user manual

This is not strictly part of the report but is a requirement for the final hand-in.

* Each method should start wide a brief description of its function.
* Use indentation to display the structure within a method.
* Comments should be used extensively. They are best used to describe logical blocks of code rather than individual statements. Line-by-line comments have the drawbacks of not providing any overview and of decreasing readability.
* Meaningful identifiers should be chosen.
* Output should be pleasingly formatted and easy to read.

# References