THE UNIVERSITY of York

**MSc, MEng and MMath Examinations, 2017–18**

**DEPARTMENT OF COMPUTER SCIENCE**

**Model-Driven Engineering (MODE)**

**Open Individual Assessment**

**Issued: Wednesday, 4th October 2017**

**Submission due: 12 noon, Wednesday, 10th January 2018**

**Feedback and marks due: 7th February 2018**

You should submit your solution through the electronic submission system (http://www.cs.york.ac.uk/student/assessment/submit/) by 12 noon on Wednesday, 10th January 2018. An assessment (or part of an assessment) that is submitted after this deadline will be marked as if it had been handed in on time, but the Board of Examiners will normally apply a lateness penalty to the whole assessment.

The feedback and mark dates are guided by departmental policy, but, in exceptional cases, there may be a delay. In these cases, all students expecting feedback will be emailed by the module owner with a revised feedback date.

Your attention is drawn to the section about Academic Misconduct in your Departmental Handbook: https://www.cs.york.ac.uk/student/handbook/

Queries on this assessment should be sent to the module leader, **Dimitris Kolovos (dimitris.kolovos@york.ac.uk)**. Answers that apply to all students will be posted on the module webpage.

**Rubric:** Your submission should include a report of no more than 5 pages, and an implementation (in EMF, GMF and Epsilon). You must submit a single zip file containing your report in PDF, and your implementation files. Your implementation files **must follow** the instructions described in Section 3.

Your exam number must be on the front cover of your assessment. You must not be otherwise identified anywhere on your submission. Do not include your name or IT services (or any other) username anywhere on your submission.

Note the page limits: parts of answers that go beyond the page limit may not be marked. Use IEEE citation and referencing: references must be listed at the end of the document and do not count towards the page limit.

# 1 Scenario

You are to design and implement a domain-specific language (DSL) for requirements modelling.

In this DSL, a requirements model comprises a number of requirements, test-cases and team members. There are two types of requirements: customer requirements and system requirements. Each requirement has an identifier and a textual description. Requirements can be hierarchically decomposed (e.g. requirement A cannot be satisfied, unless requirements B and C are satisfied) and can also be in conflict with other requirements (e.g. only one of requirements A and B can be satisfied). Each test-case has a description and can be used to verify one or more technical requirements. Some additional properties include:

- The identifier of each requirement is unique
- The textual description of a requirement is at least 10 characters long
- Customer requirements cannot be in conflict with system requirements (and vice-versa)
- Customer requirements can be decomposed into technical requirements, but not the other way round
- Each customer requirement needs to be decomposed into at least one technical requirement
- Each test-case needs to be verifying at least one technical requirement
- Each technical requirement needs originate from at least one customer requirement
- Each leaf requirement (i.e. requirement that is not further decomposed) has a "progress" field that records how much progress the development team has made towards satisfying it so far (e.g. 50%)
- The progress of a non-leaf requirement is the average of the progress of the requirements it is decomposed into (e.g. if requirement A is decomposed into requirements B and C, where B and C are at 10% and 20% complete respectively, the progress of A is 15%)
- Each requirement can be allocated to one or more team members
- Requirements decomposition must be free of cycles

Your domain-specific language will be used to achieve the following tasks:

- Construct, edit and visualise requirements models (and so you will need to provide a concrete syntax and a set of well-formedness constraints)
- Generate a static HTML web site through a model-to-text transformation that allows non-modellers to browse through the team members, requirements and test-cases of the system. In the generated website, users should be able to:
  - drill down the requirements/test-cases decomposition tree
  - visualise the progress of requirements in an intuitive way (e.g. through a progress bar)
  - view requirements assigned to particular team members
  - view requirements verified by a specific test-case
  - filter out completed requirements
- Generate a simplified requirements model through a model-to-model transformation, which excludes completed requirements as well as team members and use-cases that are only associated to such requirements.

## 2 Report

The report will discuss the following:

- **Abstract Syntax and Constraints**: Explain how you have arrived at the abstract syntax and any well-formedness constraints for your DSL, justify the decisions that you have made, and discuss how you have tested your abstract syntax and well-formedness constraints.
- **Editor**: Explain the editor for your DSL, justify the decisions that you have made, and briefly argue why your editor provides an appropriate notation for your DSL.
- **Model Management Operations**: Explain the model management operations for your DSL, justify the decisions that you have made, and briefly demonstrate that the model management operations can be used to fulfill the model management tasks described in Section 1.

Note that the report must focus on discussing how you have interpreted the problem (Section 1), on providing convincing justifications for your engineering decisions, and on convincing the examiners that your solution is sound. Therefore, the report **should not**

simply describe your solution. Instead, it should discuss your objectives, and present alternative solutions that you have considered (and describe why they were ultimately discounted). You should also describe how you tested your work, and why your testing approach was appropriate.

## 3 Submission Instructions

You must submit a single zip file containing your report in PDF and your implementation files. You must not identify yourself. For example, you must not name files with your IT services (or any other) username!

When expanded, your zip file should contain folders and files matching the structure and naming conventions described below. You must replace "Y1234567" with your own examination candidate number:

- Y1234567.pdf (your report)
- metamodel (a directory containing your metamodel files)
  - Y1234567.emf (your metamodel, in Emfatic)
  - Y1234567.evl (optional, your EVL well-formendess constraints)
  - Y1234567.etl (your model-to-model transformation)
- m2t (a directory containing your model-to-text transformation)
  - Y1234567.egl or Y1234567.egx (the entry point for your model-to-text transformation)
  - *.egl (any other templates for your model-to-text transformation)
- m2m (a directory containing your model-to-model transformation)
- models (a directory containing your models)
  - 1.model or 1.Y1234567 (your first test model)
  - 2.model or 2.Y1234567 (your second test model)
  - ...
  - N.model or N.Y1234567 (your Nth test model)

Note that:

1. Generated code (e.g. Java files generated by EuGENia) must not be included in your submission. The examiners will generate your editor during the marking process (by running the EuGENia Ant task on your Emfatic metamodel).

2. If you wish to draw attention to the execution of a model management operation on specific models (e.g., to demonstrate an interesting part of your solution), please include the sample input / output models and Eclipse launch configurations in your submission. Use the naming convention "Y1234567-Description.launch". See the section titled "Epsilon launch configurations" on the following webpage for some guidance on how to persist launch configurations: http://www.eclipse.org/epsilon/doc/articles/minimal-examples/

3. Your graphical editor should process models with an extension that matches your examination candidate number (i.e, in place of "Y1234567" in the above structure). All models that you wish for the examiners to open with your graphical editor must have this extension.

# 4 Marking Criteria

Your submission will be marked according to the following criteria:

## 4.1 Validity [15 marks]

*"Does this solution work?"*

| | 3 marks | 2 marks | 0 marks |
|---|---|---|---|
| Abstract Syntax* | The abstract syntax definition is valid Ecore. All test models conform to the abstract syntax definition. | Some minor issues with the Ecore, such as: missing statement terminators, incorrectly configured opposite references, or a few inconsistent names or types between test models and the abstract syntax definition. | The abstract syntax definition is missing, or unrecognisable as Ecore. No test models. |
| Concrete Syntax | The concrete syntax definition uses Eugenia annotations correctly. All test diagrams conform to the concrete syntax definition. | Some minor issues with the concrete syntax definition, such as misspelt Eugenia annotations or warnings encountered during code generation. | The concrete syntax definition is missing, or unrecognisable as Eugenia-annotated Ecore. No test diagrams. |
| Constraints | No syntactic errors in EVL constraints. No runtime errors occur when running the constraints on the test models. | Some minor issues with the EVL, such as: missing statement terminators, misspelt keywords, or runtime errors encountered during validation or while performing any corrective actions (fix parts). | Constraints are missing, or unrecognisable as EVL. No test models. |
| Operations | The operations are implemented as valid Epsilon programs. No runtime errors occur when running the program on the test models. | Some minor issues with the Epsilon programs, such as: missing statement terminators, misspelt keywords, runtime warnings. | Operations are missing, or unrecognisable as Epsilon programs. No test models. |

* Double-weighted.

## 4.2 Fitness for Purpose [30 marks]

*"How well does this solution address the problem?"*

| | 5 marks | 3 marks | 1 mark |
|---|---|---|---|
| Abstract Syntax | Always uses highly appropriate: representations (e.g., class vs. enum), cardinalities, reference types (e.g., containment, non-containment, bi-directional), and inheritance hierarchies. | Mostly uses appropriate representations, cardinalities, reference types and inheritance hierarchies. A few unusual and unjustified design decisions have been taken. | Confuses models and metamodels. Exhibits some of the following issues: references are used when attributes would have been more appropriate (or vice-versa), many unnecessary types and features are included, or key information cannot be described by instance models. |
| Concrete Syntax | All valid models can be constructed in the editor. Great consideration is given to the user experience (e.g., by avoiding visual ambiguity). | A few valid models cannot be constructed in the editor. | Many valid models cannot be constructed in the editor. |
| Constraints | Precisely distinguish valid and invalid models. No redundant constraints. Some steps are taken to make it easier for users to fix problems. | Distinguish valid and invalid models. Some constraints are redundant (e.g., are repetitive of the abstract syntax definition). | Every constraint is either redundant, or does not distinguish between valid and invalid models. |
| Operations | Produces highly appropriate output from all valid input models. Considers the way in which the outputs will be used. | Produces appropriate output from most valid input models. | For many valid input models, either no output is produced or the output is inappropriate. |
| Overall | Permits the construction of all reasonable valid models, and produces appropriate outputs from every reasonable and valid model. Invalidity is detected as early as possible (e.g., few invalid models are permitted by the metamodel defintion). | Permits the construction of most reasonable valid models, and mostly produces appropriate outputs from every reasonable and valid model. A few valid models are not accepted, or a few invalid models are not disallowed. | Many valid models are not accepted. Many invalid models are not disallowed. |

### 4.3 Habitability [15 marks]

*"How easy would it be (for other MDE developers) to change this solution?"*

| | 3 marks | 2 marks | 1 mark |
|---|---|---|---|
| Cohesion* | The solution is organised and structured so that its parts work in harmony. For example, the abstract syntax is structured to support the implementation of the concrete syntax and the operations. | Occasionally, some parts of the solution must compensate for a deficiency in others. For example, an operation must perform a complicated query due to the structure of the abstract syntax. | Often many parts of the solution must compensate for deficiencies in others. For example, operations must perform several complicated queries due to the structure of the abstract syntax. |
| Names | Names (for types, rules, variables, etc.) are clear, and often taken from the problem domain, or from the solution domain. Uses consistent naming conventions. | Most names are clear and easy to understand. Uses consistent naming conventions. | Most names are clear and easy to understand. |
| Patterns and idioms | Highly appropriate design patterns and MDE idioms have been applied. | Occasionally, the use of design patterns or MDE idioms would have reduced implementation complexity. | In several places, the use of design patterns or MDE idioms would have substantially reduced implementation complexity. |
| Evolution | There is some evidence that thought has been given to likely future changes in requirements. | The design supports likely future changes, though possibly unintentionally. | The design is brittle and would be difficult to change. |

* Double-weighted.

### 4.4 Mastery [10 marks]

*"Does this solution demonstrate advanced MDE techniques and independent learning?"*

Marks are awarded for using advanced features of EMF, GMF, Epsilon or Eugenia when doing so improves the solution. For inspiration, students are encouraged to explore the Epsilon website: https://www.eclipse.org/epsilon/

### 4.5 Justification [30 marks]

*"How defensible is this solution? Have trade-offs been identified and alternatives considered?"*

| | 5 marks | 3 marks | 0 marks |
|---|---|---|---|
| Approach | Provides a clear and concise description of how the problem has been solved. Each part of the solution is justifiable and it is clear that the reasoning behind each MDE artefact is well understood. | Discusses how the problem has been solved, and how each part of the solution contributes to solving the problem. | No useful description of how the problem has been solved, or of how each part of the solution contributes to solving the problem. |
| Analysis* | Clearly articulates the challenges associated with solving the problem. Contextualises challenges with respect to the environment (e.g., university coursework with a fixed timescale). | Articulates some of the major challenges associated with solving the problem. | No challenges are identified. |
| Decisions* | Reasonable alternative designs for every part of the solution are presented. Highly appropriate reasons are given for choosing one design over another. Decisions are related to the challenges identified during analysis. | Either alternative designs are not presented for several parts of the solution, or many alternative designs are largely superficial. | No alternative designs are considered. |
| Evidence | Every decision is supported by a convincing argument. Where appropriate, other documentary evidence is provided (such as screenshots). | Every decision is supported by a reasonably convincing argument. | Decisions are not supported by any evidence. |

* Double-weighted.