WATCHME

Workplace-based e-Assessment Technology for Competency-based Higher Multi-professional Education

A COMPREHENSIVE MANUAL FOR IMPLEMENTING THE WATCHME TOOLS

Dr. Jeroen Donkers

Maasticht University

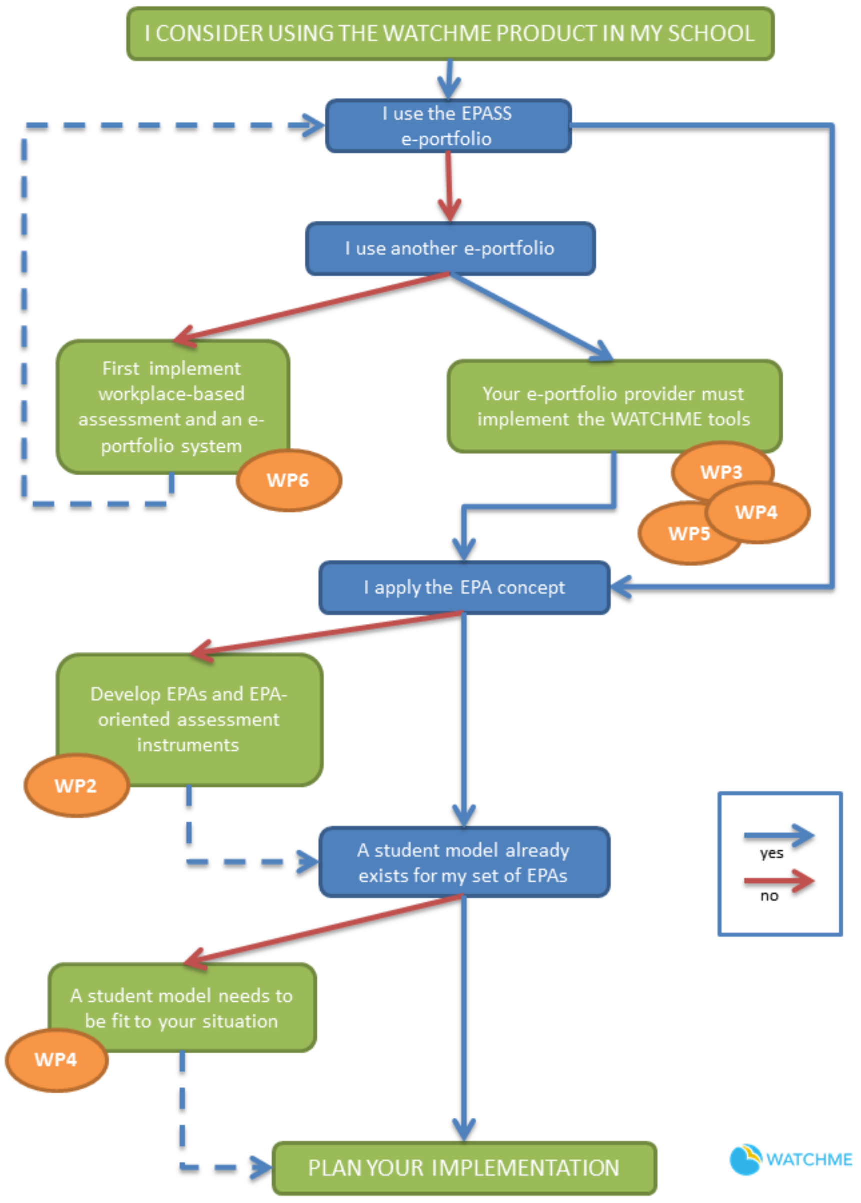
Jeroen.donkers@maastrichtuniversity.nl

www.project-watchme.eu

This project has received funding from the European Union's Seventh Framework Programme for research, technological development and demonstration under grant agreement no 619349.

1 Introduction

When you plan to apply the WATCHME tool in your own school, the following flowchart will guide you to the work packages and deliverables of the project you need to study.


I CONSIDER USING THE WATCHME PRODUCT IN MY SCHOOL
I use the EPASS e-portfolio
I use another e-portfolio
First implement workplace-based assessment and an e-portfolio system
WP6
Your e-portfolio provider must implement the WATCHME tools
WP3
WP4
WP5
I apply the EPA concept
Develop EPAs and EPA-oriented assessment instruments
WP2
A student model already exists for my set of EPAs
A student model needs to be fit to your situation
WP4
PLAN YOUR IMPLEMENTATION
yes
no
WATCHME

In this technical manual we collected some selected chapters from deliverables that might help in implementing and adjusting the WATCHME software tools. There are three chapters. In the first chapter we explain how to design a software architecture for using student models in your own (e-portfolio) system. The second chapter explains how to build a simple student model using the MEBN/PrOWL2 tool in UnBBayes. The last chapter explains the system requirements and procedures to use when you decide to implement the original WATCHME system as available in the github respository (www.github.com/watchme-opensource)

2 A compact incorporation of the Watchme Student model into EPASS (or any other e-portfolio system)

(Where EPASS is mentioned in this chapter, any other e-portfolio systems can be understood as well. This chapter is part of Deliverable 7.4)

The complex architecture (deliverable 3.2, and attachment A of deliverable 4.3) that was developed in the WATCHME project was given in by the demands that

- a separate server would be used for WATCHME purposes
- the WATCHME system could be connected to other e-portfolio systems
- quick exchange of information between EPASS and WATCHME was needed since visualisations also use the SM module for intermediate storage and preprocessing
- Several technical partners with different backgrounds needed to cooperate closely to develop the system

The disadvantages of the current architecture that would hinder further exploitations in EPASS are:

- maintenance of an additional server, running tomcat, mongodb, natural language processors and other software that needs to be kept up-to-date and for which insufficient expertise is present in the EPASS team
- need for secure and complex APIs between separated system parts, even for simple tasks
- a separate storage of portfolio data in a non-structured database on the WATCHME server, leading to additional security risks and maintenance issues
- an elaborate configuration using sets of additional config files
- Parts of the model processing are hard-coded in java code and not generic
- The current high frequency of data processing is not in line with the slow pace of changes in workplace-based learning.

However, when student modelling is incorporated into EPASS, these demands are no longer valid. It means that:

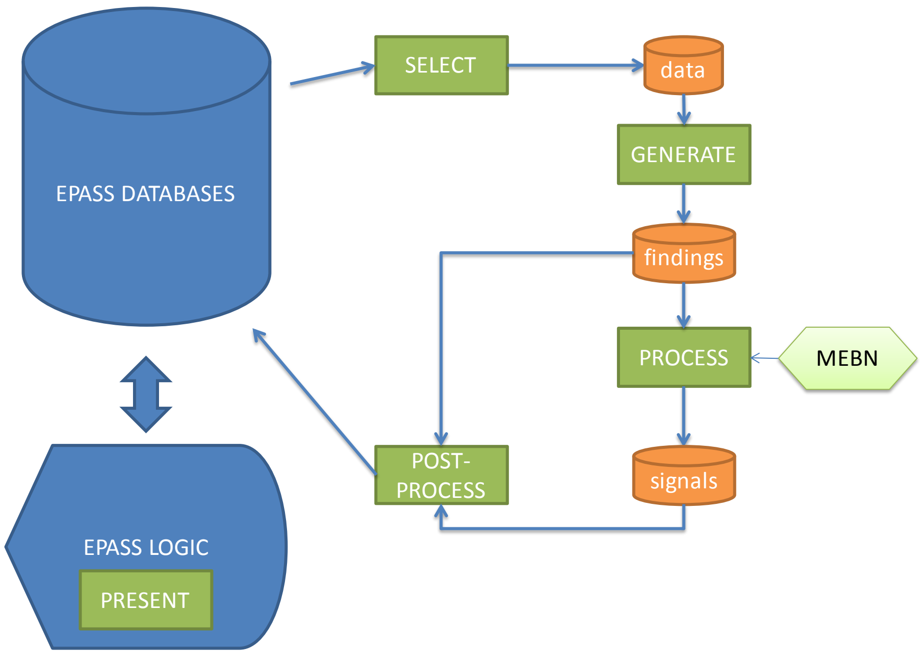
- A far less complex architecture is needed
- The EPASS databases can remain the only storage place of portfolio data and EPASS configuration can take care of configurations needed for student models and feedback messages
- Computations needed for student modelling can be done on EPASS servers, probably overnight, on a weekly basis

2.1 A more compact architecture for student modelling in EPASS

The processing pipeline of EPASS portfolio data for student modelling involves the following steps (in terms of information flow):

- SELECT: Extraction of relevant portfolio information (into a temporary data file). This involves a certain time window of latest portfolio data.

- GENERATE: Pre-processing of the data to detect findings, stored in a second data file (these findings will also be stored in the EPASS database, and can be used for analytics and feedback). In future, this step can incorporate natural language processing.
- PROCESS: Processing of the findings by MEBN student models. This will result in signals (probabilities per output node) that will be stored into the EPASS database.
- POSTPROCESS: Storing of data in EPASS database and if needed, aggregations of findings and signals can be computed to be stored in the EPASS database for faster performance
- PRESENT: Internal EPASS logic will translate the findings and signals into JIT messages on student and supervisor pages and into visualisations



Processing pipeline for student models in EPASS.

The platforms for each step will differ:

- SELECT: SQL query, Perl/Python script, or PHP module
- GENERATE: Java (at the moment), Perl, PHP
- PROCESS: must be done in Java, due to the available MEBN libraries
- POSTPROCESS: PHP or Perl/Python
- PRESENT: PHP and Javascript

All of these platforms can be made available at the EPASS server and can be scheduled for automated run using a cronjob and Perl/Python scripts.

2.2 Design decisions to take, part 1: processing

These steps are depending on each other and on the needs of the domain to which the student model is applied. The configuration of these steps needs to be carried out in a given sequence.

1. Design of student model MEBN: this determines the findings needed for input and the signals that are produced
2. Selection and implementation of methods to generate findings
3. Definition of data extraction queries
4. Design of post-processing procedures
5. Configuration / design of presentation logic

Time scale

An important design decision is the time scale or granularity at which the student model should operate. In many cases, this will be on the scale of weeks, but it is thinkable that a scale in months or days will be more convenient in some domains.

Table design

A prerequisite is a proper and generic table design in EPASS to store findings and signals. Some preliminary ideas:

Table design for findings:

1. Finding id
2. Student id
3. Date   (could also be a time point, e.g. week)
4. Name of the finding type (either a string or a finding type id)
5. Parameters of the finding (e.g., epa, competence)
6. Value of the finding (either a string or a value id)

(Columns 2-5 unique)

Table design for signals:

1. Signal id
2. Student id
7. Date (could also be a time point, e.g. week)
3. Name of the signal type (either a string or a signal type id)
4. Parameters of the signal (e.g., epa, competence)
5. Value (numeric between 0 and 1)

(Columns 2-5 unique)

The parameter column can contain more than one value, this number is fixed for each finding or signal. This could be normalized further, or just processed by logic. If ids are used for finding types, signal types and value ids, these could be stored in single table.

Table design for types:

1. Typeid
2. Type name
3. Type class (finding, signal, value)
4. Number of parameters

Generating findings

The student model design dictates what findings are needed for input. In the current WATCHME model, these findings are generated from sequences of assessment scores and their dates. Statistical procedures are used to decide for the presence of absence of signals such as a sudden drop in score. At the moment these finding-generating functions are developed in Java, using a statistical toolkit. These functions take as input arrays of scores and/or time points. These time points are discrete dates, rounded of the the desired time scale (in this case: weeks). Time points should be relative to “now”. They either could start at zero at the starting point of the overall time window, or start with zero at “now” and run into negative numbers for the past.

The functions need data from a certain time window *w* in order to be able to make decisions. So if for a given time point *t* findings need to be generated, data from window [*t*-*w*, *t*] is needed. However, since the student models are dynamic (they reason over a period of time, called the depth *d*), data from period *(d+w)* before the current time is needed. The current value for *d* is eight weeks, and *w* is 24 to 32 weeks.

The scores that are needed for the findings, might not be directly available in the EPASS database. In the case of the veterinary medicine test case, competency scores were used in the model. These scores (per assessment) are already pre-computed and stored in an EPASS table. When other scores are needed (e.g. EPA scores), they have to be computed either:

- In a new table in the EPASS database, like the form-scores table
- During the export of EPASS data
- During processing of the exported data by the finding generator

Currently, the last option is used. The finding generator uses a set of data structures that store and processes the assessment data per student in memory and allows easy generation of the arrays needed for the finding generating functions.

The output of the finding generator is a temporary file containing on each line a finding for one student for one time point. The naming of the findings, parameters and values should match those in the MEBN network of the student model.

Extracting data from EPASS

The finding generator currently uses a comma-separated CSV file which either one line per assessment per competency, or one line per question (depending whether form-scores or individual answers are extracted). In the latter case, a separate mapping file between question and EPA is used. The query was complicated since students and assessment types had to be selected according to whether they were involved in WATCHME.

In the ideal case, the EPASS database should contain all information to be able to extract the necessary scores, either per assessment or per question. A single query (stored procedure) should be sufficient to collect all data, using only a single parameter for the overall time window size. Conversion of dates into time points can be part of that query.

The result of the query should be stored in a temporary file accessible to the finding generator.

Post processing

If the table definitions are in place, the output of the finding generator and the student model can be translated easily into insert queries. It might be worthwhile to consider allowing updates of earlier values. Since especially findings are sparse, an update for findings for a given time point *t*, should consist of first removing all findings for that student for time point *t* and then inserting the new ones.

The relative time points need to be translated back to true dates (or absolute time points) before data is stored in the database. Names for findings, parameters and values might also need some post formatting. Translation tables might be needed for this last task, but preferably some fixed translation rules should be used.

The post processing can be done in PHP, Perl or Python, as long as easy SQL/Database access is possible.

Student model processor

The student model processor takes the output of the finding generator and produces a file with on every line a separate signal per student per time point.

The processor collects all findings for a student in the required depth (e.g. 8 weeks) and feeds those as evidence into the MEBN model. It then queries the output nodes of the MEBN and reads out the resulting probabilities for the current time. It appends those probabilities to the temporary signals file.

The student model processor must be developed in Java since it needs to use the UnBbayes toolkit to process the MEBN files. It can be compiled into a single executable JAR file, that only needs a few command line parameters to connect findings file to the appropriate MEBN files. The current module used in WATCHME is almost in that stage and needs only little adaptations. To select the appropriate output nodes for signals, a separate configuration file is needed that lists these nodes. (Maybe some way can be found to mark those nodes inside the MEBN fragments directly).

This processor must be run for relatively small batches since the current version of the unbbayes library contains a memory leak. Moreover, error messages must be suppressed, since it might generate too many of them.

Synchronization

The above steps should be performed one after the other and wait for output. This probably can be arranged in a shell script, or a Perl/Python script.

Logging and error handling

Since the above steps will run unattended, sound logging is needed. However, since different platforms are used, we cannot use a single approach. At the start, the temporary files should be compressed, time-stamped and stored in a repository directory. Later this can be omitted. All steps should append a line in a common log file with start time and stop time. A separate error log should be kept for unexpected situations. An error in one step should abort the whole process.

2.3 Design decisions to take 2: Presentation

There are a few possible ways to use the signals and findings tables in the EPASS database.

Messages

Just like the JIT module currently developed in WATCHME, the probabilities in the signals table can be used to show messages to students and supervisors. To do this, three things are needed:

1. Decide on what pages messages should be provided for which user
2. For each page, define rules that dictate when messages are displayed
3. Design messages to be displayed when a rule fires

A single message-displaying widget can probably be reused for all pages were the message is needed.   A single procedure can be developed that generates the messages. The procedure would take four parameters: the student id, the focus time point, a code for the page, possible context parameters (e.g. EPA, competency)

A database table (or a config file) containing the rules is needed as well. Based on the parameters that are provided, first the set of appropriate rules is selected. Then the signals and findings for the student and the focus time point are extracted from the database. Third, the rules are one-by-one processed on the extracted data set and if they fire, a message is generated, using a message template and the parameters.

Pages for messages can be: opening screen, an assessment form, an evaluation form, a general progress page, and so on.

For supervisors, the messages for several students could be combined in a single list of messages for students involved.

For the rules, a general rule format needs to be designed, depending on the desired complexity of the rules. Here we have a choice of putting the complexity inside the student model, so signals are easily interpreted, or allowing more complexity in the rules, such as combining multiple signals in a rule. In the simplest case, rules can be represented by a single table:

- Rule id
- Place code (to which page does the rule apply, maybe also for groups of pages, or ‘all pages’. The code also incorporates the user type)
- Signal/finding id
- Level (= 0 or 1 in case of a finding, else a probability such as 0.5)
- Message template id

In the case of context (EPA, competency), the parameters of the procedure are used to select the findings/signals with the appropriate context parameters.

The message templates are strings with placeholders for student names, or context parameters. Each message template can have multiple variants, from which one is selected by chance. The templates are also language dependent (localisation).

Tables

It is also possible to present a table that lists the current signals in numerical form (percentages). This functionality is not present in WATCHME. This information can be derived directly from the signals table. The name of the signals then need to be translated into human-friendly and localized phrases.

The tables can use parameters to make them context sensitive (e.g. per EPA). They also could contain the signals for multiple students, in case of a supervisor user.

Preferably, a table widget should be developed that takes as parameter the names of the signals it should present.

Visualisations

Currently in WATCHME only EPA scores (in fact, some kind of score estimations) are presented. It is possible to present findings and signals as well graphically. Maybe not all signals and findings are presentable to all users, and contextualization might also be wanted (similar to the tables above).

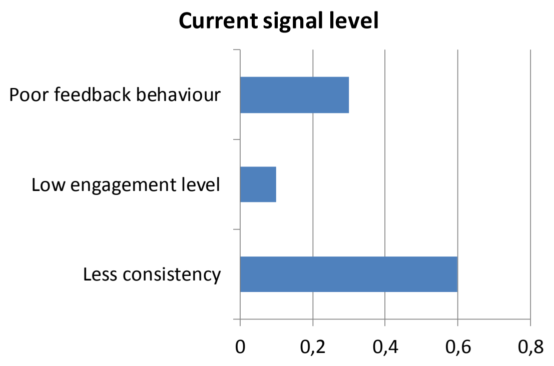
Visualisations for signals can be:

- Single point of time, for multiple signals, or contexts
  - Bar chart, spider chart, colour-augmented table
- Longitudinal, either one signal or multiple signals
  - Line graph
- Longitudinal graph for multiple students
  - Line graph

Findings can also be visualized, although they only have a few discrete levels, using dot plots.

Examples for visualisations

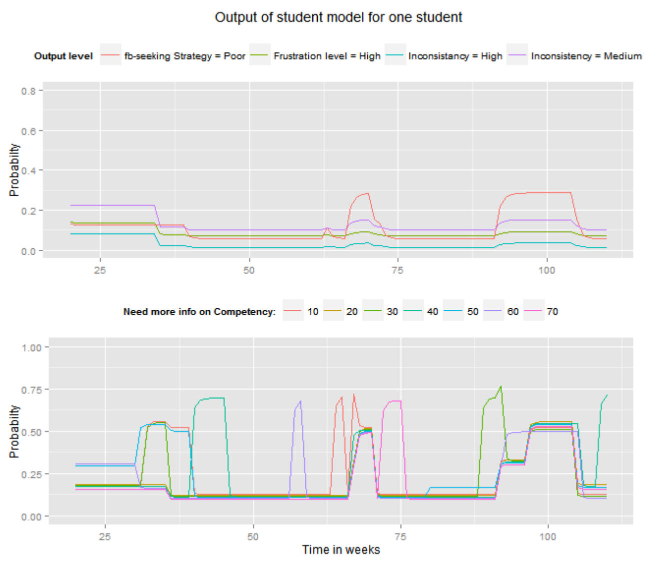
A bar chart:

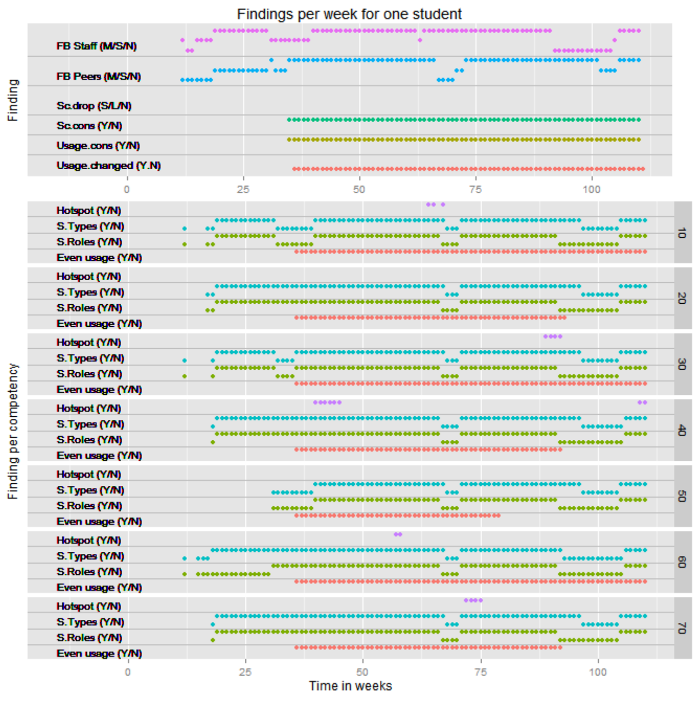


A coloured table:

Poor feedback behaviour	0.3
Low engagement level	0.1
Less consistency	0.6

Signal graph:





The graph-widgets should also be generic, in such way that parameters can be set to which signals and findings are presented. Ultimately, a dashboard could be developed in which the user can select the proper visualisations. (Including all EPASS visualisations).

2.4 Configuration of student models for an application domain

What steps need to be taken to configure a student model for an application domain? These are the non-generic parts of applying the student model.

A) Develop the student model and finding generator

The student model itself consists of knowledge fragments that are in principle reusable. Some ideas are very general and might be applicable to many domains. Still, detailed tuning of probabilities will be necessary on the basis of particular properties of the domain.

The same holds for the finding generator. The functions used to distillate the findings might need tuning to the particular properties of the domain.

B) Properties for the data extractor

Probably the data extractor will not vary much between application domains. However, this depends on whether individual question answers need to be extracted or whether aggregated form scores are available. Also the time span needed for extraction needs to be set.

C) Properties for model processor

The model processor needs to know the following:

- Which model to use
- What depth (number of time steps)
- What output nodes (and states) it should query

The names of the findings should match the name of the input nodes of the model, so no configuration should be needed for that.

D) Design rules and message templates for messages

This is very domain specific: where and when are messages needed? This includes deciding where the message widgets are placed on the pages.

E) Set up visualisations

This includes selecting and placing the widgets and setting parameters.

F) Postprocessing

If rules are used for translation of signal and finding names, there is no need for configuration for this processing step.

3 Building a student model using MEBN/PrOWL2

This chapter is a hands-on manual on how to build a student model using MEBN/PrOWL2. It has been developed as part of a WATCHME workshop. The software that you need to install for this chapter is:

- Java JRE (recent version)
- Protégé ontology editor 5.0 or higher (http://protege.stanford.edu)
- UnBBayes (http://sourceforge.net/projects/unbbayes)
    - GUI and plugins unbbayes.prs.mebn, unbbayes.gui.mebn.ontology.protege
    - Versions: Gui: 4.21.15, plugins 1.13.11 and 1.1.4, *or*
      Gui: 4.21.18, plugins 1.14.13 and 1.2.5.
- Some Java IDE (e.g. Eclipse) (JDK needed)

We are going to create a very simple student model (with two MFrags).

3.1 Part 1: setting up your ontology

1a. create empty ontology file

- Start Protégé.
- Rename the ontology (field Ontology IRI) to: “studentmodel”.
- Save the model in OWL/XML format under the filename “studentmodel.owl”

1b. create Classes

- Switch to tab “Classes”
- Create classes Time, Task, Skill (all three, subclasses of Thing)

1c. create Instances (individuals)

- Switch to tab “individuals”
- Create individuals for Skills: “writing”, “desinging”, “programming”.
- Create individuals for Tasks: “task_1”, “task_2”
- Save ontology

1d. create object properties (relations)

- Switch to tab “object properties”
- Add property “needsSkill” (under topObjectProperty)
- Select “Task” for the domain and “Skill” for the range of this object property.
- Save ontology

1e. create data properties

- Switch to tab “data properties”
- Add property “finishedTask”
- Add as domain: Task and Time
- Select as range the built-in data type “xsd:Boolean”
- Add properties “practicedSkill” and “hasSkill”
- Add as domain: Skill and Time to both
- Select as range the built-in data type “xsd:Boolean” for both
- Save ontology

1f. create relation between instances

- Switch to tab “individuals”
- Select “task_1”
- Add object property assertion: needsSkill designing
- Add object property assertion: needsSkill writing
- Select “task_2”
- Add object property assertion: needsSkill programming
- Add object property assertion: needsSkill writing
- Save ontology
- Close Protégé

3.2 Part 2: creating the MEBN

2a. create an MEBN file pair (.ubf + .owl)

- Open UnBBayes
- Open file studentmodel.owl form part 1
    - Select file type “MEBN with PR-OWL 2.0”
- Save the file under name “studentmodel.ubf”
    - Select file type “MEBN with PR-OWL 2.0” (**DO THIS ALL THE TIME!!!)**
- Close UnBBayes
- You will find that two linked files have been created:
    - Studentmodel.owl
    - Studentmodel.ubf
- Open studentmodel.owl in Protégé and observe the additions (imported ontology “pr-owl2.owl”) DO NOT SAVE!

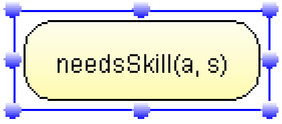
2b. add time steps

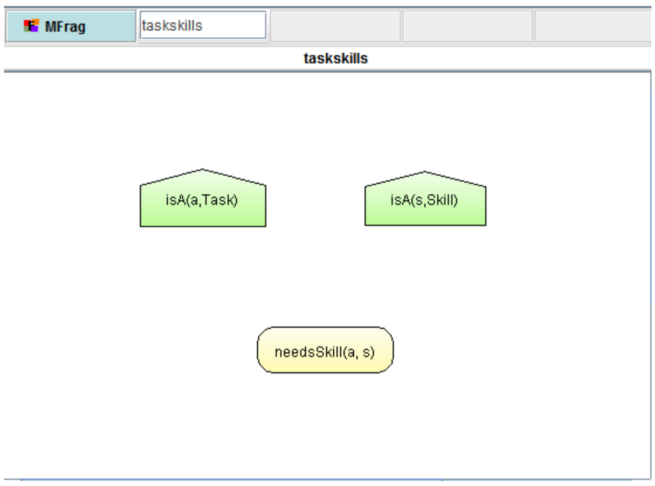
- Open UnBBayes
- Open file studentmodel.**ubf** form part 2a (**DO NOT SELECT THE OWL FILE)**
    - Select file type “MEBN with PR-OWL 2.0”
- In the MTheory tab, select MTheory entities pane (button with the orange circle):
- Select Class “Time” and check the box “is Ordenable”
- Select the Entity instances pane (diamond and orange circle):
- Select “Time [Ord]”
- Add time instances “T0”, “T1”, “T2”, “T3” (enter name and use “+” button).
- Save the file (Select file type “MEBN with PR-OWL 2.0”)

2c. create the MFrag “taskskills”

- Insert a new MFrag (use third vertical button):
- Change the name of the MFrag in “taskskills” (name field at top of right pane, press Enter)
- Insert a task variable
    - Press “insert Ordinary variable” button
    - Click in the right pain to place the variable node
    - Select “Task” from the dropdown box

    - Enter “a” as the name for the variable
    - Enlarge the “IsA” node symbol to make the text is readable:
- Add a variable “s” of type Skill
- Add the relation node “needsSkill”
    - Select the “Property2Node” tab on the top
    - Select the middle button (“Show OWL properties”)
    - Locate property “needsSkill” and drag it to the right pane
    - Enlarge the node to make the text readable.
    - Select the “Mtheory” tab on de top
    - Unselect and reselect the “needsSkill” node (the left pane will change
    - Select the button for parameters (parantheses)
    - Double click on “a (Task)” and double click on “s (Skill)” to add the pa
    - Click on the “T+” button and the “+” button to set Boolean values for this node (true, false, absurd).
- Save your file

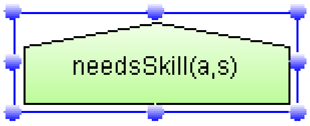




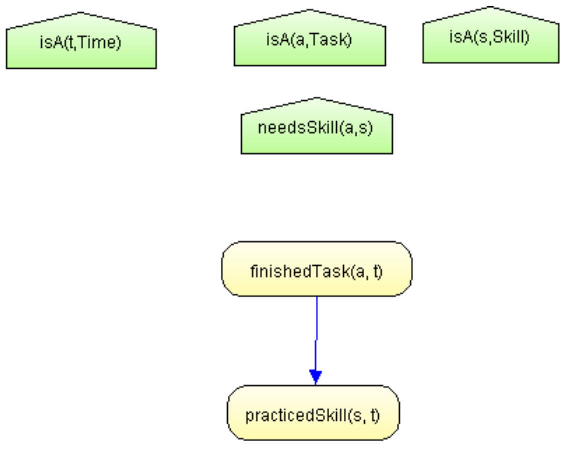
The result of 2c: MFrag taskskills

2d. create the MFrag “practice”

- Insert a new MFrag “pratice”
- Insert variable t of type Time
- Add a variable “a” of type Task and a variable “s” of type Skill
- To restrict the relation between task and skill:
    - Add a context node to the Mfrag (use button “C”):
    - Double click “formula” in the left upper pane
    - Double click node “needsSkill” in the left lower pane (tree)
    - Double click on “needsSkill” that just appeared in the upper pane
    - Select “a” for the Task_label and “s” for the Skill_label
    - The node should now look like:
- From the Property2Node tab drag property “finishedTask” to the MFrag.
    - Set parameters to “a” and “t”
    - Add Boolean values to this node



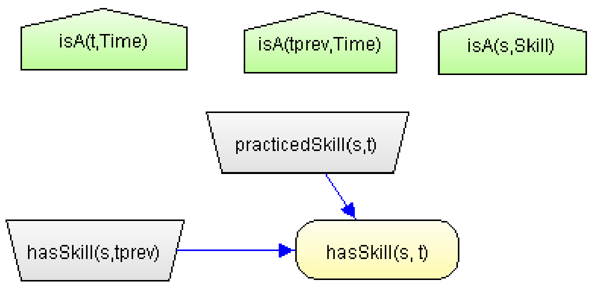
- From the Property2Node tab drag property “practicedSkill” to the MFrag.
    - Set parameters to “s” and “t”
    - Add Boolean values to this node
- Add an arrow from finishedTask to practicedSkill:
- Save your file



Result of 2d: MFrag practice

2e. create MFrag “skilldevelopment”

- Insert a new MFrag “skilldevelopment”
- Insert variables t and tprev of type Time
- Add a variable “s” of type Skill
- From the Property2Node tab drag property “hasSkill” to the MFrag.
    - Set parameters to “s” and “t”
    - Add Boolean values to this node
- Add an input node using the “I” button
    - Select node “hasSkill” from the left tree
    - Select “s” for the Skill_label
    - Select “tprev” for the Time_label
- Add an arrow from inputnode “hasSkill(s,tprev)” to “hasSkill(s,t)”
- Add a second input node
    - Select node “practicedSkill from the left tree
    - Select “s” and “t” for the labels
- Add an arrow from inputnode “practicedSkill(s,t)” to “hasSkill(s,t)”
- Save your file



The result of 2e: MFrag skilldevelopment

2f create local distributions

- Open MFrag Practice, click on node “practicedSkill”
- Open the probability table editor:
- Enter the following formula (using the buttons):

```
 if any t have ( finishedTask = true)   [
         true = 0.8,
         false = 0.2,
         absurd = 0
       ] else [
         true = 0.2,
         false = 0.8,
         absurd = 0
 ]
```

- Click on “Save” and then on “Compile”
- In node “finishedTask” add the formula:

```
[
         true = 0.01,
         false = 0.99,
         absurd = 0
]
```

- Remember to press “Save” and then “Compile”
- Open MFrag skilldevelopment, click on “hasSkill”
- Edit the probability table and fill in the following formula:

```
if any t have ( practicedSkill=true ) [
      if any tprev have (hasSkill=true) [
         true = 0.9,
         false = 0.1,
         absurd = 0
      ] else [
         true = 0.7,
         false = 0.3,
         absurd = 0
      ]] else [if any tprev have (hasSkill=true) [
         true = 0.6,
         false = 0.4,
         absurd = 0
      ] else [
         true = 0.1,
         false = 0.9,
         absurd = 0
]]
```

- Remember to press “Save” and then “Compile”
- Save your file

3.3 Part 3: querying the network

Now we are ready to query the MEBN network.

First add some context knowledge:

- Open the findings editing pane
- Select needsskill and use the pencil button to add
  - Task 1 needs skill writing
  - Task 2 needs designing

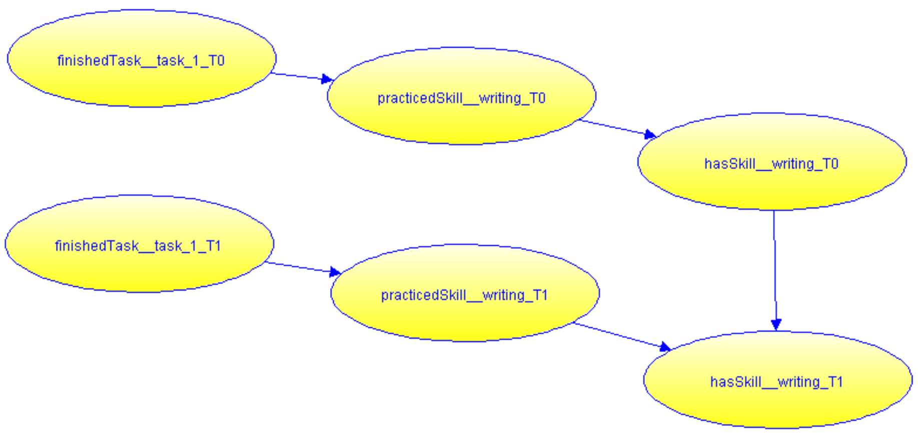
Then add an observation:

- Select finishedTask and use the pencil to add:
  - Task 1 was finished on T0

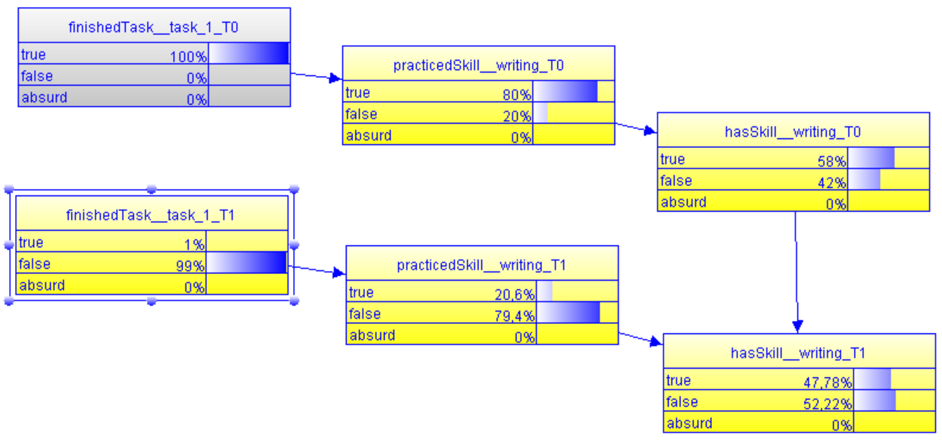
Now we can query:

- Press the query button:
- Select “hasSkill” from the list
- Select skill “writing” and time “T1”
- Press Execute

The following query-graph (SSBN) appears (after rearranging the nodes a bit):



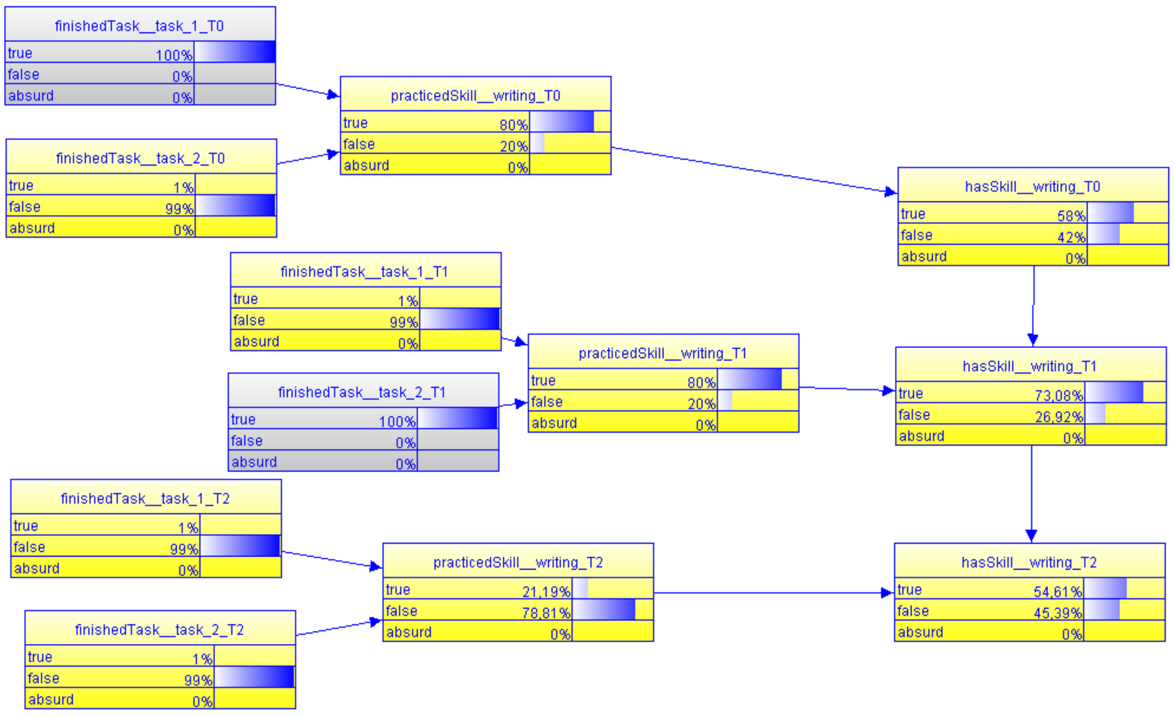
On the left you can see the resulting properties. By right-clicking on the nodes, you can show the belief bars in the nodes:



You can see that the skill decreases a bit, because it was practiced only at T0.

Use the edit network button to return to the edit mode:

Now change the context such that Task 2 also needs writing skill. Add the observation that task 2 was finished at T1. Query the writing skill at time T2.



3.4 Part 4: querying the network in Java

Import the eclipse project into Eclipse and open the “main.java” file:

```
public static void main(String[] args) {

File ubf = new File("C:\\\\gala_workshop\\\\models\\\\studentmodel.ubf");
try {
    UnBBayesWrapper ub = new UnBBayesWrapper(ubf);
    ProbabilisticNetwork pr=null;

    // add context knowledge
    ub.addBooleanEntityFinding("needsSkill", new String[] {"task_1", "writing"},true);
    ub.addBooleanEntityFinding("needsSkill", new String[] {"task_2", "writing"},true);

    // add observations
    ub.addBooleanEntityFinding("finishedTask", new String[] {"task_1", "T0"},true);
    ub.addBooleanEntityFinding("finishedTask", new String[] {"task_2", "T1"},true);

    // make query
    List<QueryItem> q = Arrays.asList(new QueryItem("hasSkill",
                                Arrays.asList("writing", "T2")));

    // run query
    pr = ub.query(q);

    // collect output probabilities
    System.out.println("OUTPUT:");

    ProbabilisticNode pn = (ProbabilisticNode) pr.getNode("hasSkill__writing_T0");
    System.out.println("p(has writing skill at T0) = "+pn.getMarginalAt(0));

    pn = (ProbabilisticNode) pr.getNode("hasSkill__writing_T1");
    System.out.println("p(has writing skill at T1) = "+pn.getMarginalAt(0));

    pn = (ProbabilisticNode) pr.getNode("hasSkill__writing_T2");
    System.out.println("p(has writing skill at T2) = "+pn.getMarginalAt(0));


} catch (Exception e) {
     System.out.println(e.toString());
     e.printStackTrace();
}}
```

5 Deployment requirements and procedures as applied in the WATCHME project

This chapter from Deliverable 4.4 shows the requirements and procedures needed to deploy the WATCHME system as it is produced in the project. The source code for the tools is available at: www.github.com/watchme-opensource

5.1 System Requirements

Software	Version	Observations
Oracle Java Virtual Machine	≥ 1.8	The Open Source JVM may work fine as well, but the whole deployment has been tested with Oracle version.
Apache Tomcat	8.x	
MongoDB	2.6.x	Please note that version 3.x is not supported at the moment by the SM.
Apache Maven	≥ 3.1.1	For deployment only
Robomongo	≥ 0.8.5	**(Optional)** For database administration. Alternatively, the mongo command line tool can be used

Hardware	Specifications
Processor	Dual Core @ 2.0 GHz or more
RAM	1 Gb or more
Memory allocated to Tomcat[1]	512 Mb or more
Hard disk (free space)	10 Gb or more
Network	Any

5.2 Project System Setup

Software	Version
Ubuntu Linux	15.04 non-LTS
Oracle Java Virtual Machine	1.8.0_45-b14
Apache Tomcat	8.0.14

[1] The memory allocated to the whole system needs to be higher than the memory allocated to Tomcat.

MongoDB	2.6.10

Hardware	Specifications
Processor	Quad Core @ 2.0 GHz
RAM	12 Gb
Tomcat Memory	8 Gb
Hard disk (free space)	200 Gb
Network	≥ 900 mb/s

5.3 Deployment Procedure

The following steps are to be done in order when deploying a fresh instance of the Student Model.

5.3.1 Database setup

This step assumes that MongoDB has been fully deployed and it is functional.

The only database setup that needs to be done is to create the Student Model database.

Using the MongoDB command line interface issue the following command:

```
use DatabaseName;
```

5.3.2 Build the Student Model

The Student Model comes with a few automatic build scripts, for Windows and Linux alike.

Typically, there are two build scripts per target environment – one for each supported platform: Windows and Linux. Additionally, there are two clean-up scripts, one for each supported platform.

At the time of this writing the supported target environments for Student Model are:

Environment	Description
Local Development (dev)	The development environment. This can vary with each Student Model developer machine. Build files: - build-dev.bat (Windows) - build-dev.sh (Linux)
Shared Development (development)	The shared development environment, hosted on the production server, at http://server.project-watchme.eu/sm-development/sm/api. Build files: - build-development.bat (Windows) - build-development.sh (Linux)

Test (test)	The acceptance environment, hosted on the production server, at http://server.project-watchme.eu/sm-test/sm/api. Build files: - build-test.bat (Windows) - build-test.sh (Linux)
Production (prod)	The production environment, hosted on the production server, at http://server.project-watchme.eu/sm-test/sm/api. Build files: - build-test.bat (Windows) - build-test.sh (Linux)

To build the Student Model first the build scripts have to be located and then a target environment needs to be chosen.

The build scripts are located in the root folder of the project. The output will be placed in *_build_/${war_name}*, where *${war_name}* is the environment name prefixed with *"sm-"* (for example *sm-test*).

The build output consists of two artifacts, both placed in *_build_/${war_name}*:

- The exploded WAR file, stored in a directory called *explodedwar*.
- The actual WAR file, named *sm-${env_name}.war.*

Either artifact can be deployed to Tomcat.

It is recommended that after a successful build and prior to a deployment the application configuration file be examined and reviewed.

After a build, the file is placed in *webapi/api/src/main/webapp/WEB-INF/appsettings.properties.*

5.3.3 Configuration for target environment

This step should be checked and performed pre-build.

The Student Model build system supports and requires a configurable section per build profile (as in dev, development, test or prod). This means that the configuration settings are different per profile.

The pre-build settings are split in two files:

- A set of configurable properties that sit in *webapi/api/profiles/${env_name}/config.properties*.
- A static set of properties which sit in *webapi/api/src/main/webapp/WEB-INF/appsettings.properties.common*.

Only the first file has to be customized pre-build by the deployer. The settings categories that need to be customized per environment are:

- General server settings,
- EPASS Privacy Manager credentials, per domain,
- EPASS URLs,
- MongoDB settings,
- Domain-related settings,

- Domain files paths,
- Static files paths
- JIT and VIZ caching preferences
- Logging options
- Various parameters for the Bayesian network.

5.3.4 Full Student Model deployment

A full Student Model deployment implies overwriting an existing version deployed in Tomcat with a new version.

The deployment can be done effectively using any means that are available to the deployer: Tomcat Manager web interface, SSH interface/command line, SCP or similar protocols or tools.

5.3.5 Partial Student Model deployment

A partial Student Model deployment implies either a configuration change, in the runtime configuration file or changing only a subset of the Student Model runtime – which can be either classes (not recommended) or static files (more likely to occur in practice).

The steps to make a partial deployment are straightforward:

- Copy the new runtime files to the server where Tomcat is running.
- Copy the new runtime files over the existing ones, in the Tomcat deployment directory for Student Model.

Restart the web application via the Tomcat Manager web interface or tomcat-manager command line tool that is bundled with Tomcat.