ECSE-539 Advanced Software Engineering
Winter 2022

Autonomous Prognostics and Health Management Automation Language (APHMAL)

Project Report

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081

April 10, 2022

ECSE-539 Advanced Software Engineering
Autonomous Prognostics and Health Management Automation Language (APHMAL)
Project (April 10, 2022)

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081
(the alphabetical order reflects also the contribution made by the team members in descending order)

TABLE OF CONTENTS

ECSE-539 Advanced Software Engineering
Autonomous Prognostics and Health Management Automation Language (APHMAL)
Project (April 10, 2022)

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081
(the alphabetical order reflects also the contribution made by the team members in descending order)

1. Introduction

The Prognostics and Health Management (PHM) is a part of Systems Engineering domain and is essential to properly design and implement aerospace-related programs and projects while PHM for Human Health and Performance (HH&P) is a PHM-based application, since the paradigm shift on the programs from telemedicine to HH&P autonomy and NASA's adoption of Model-Based Systems Engineering (MBSE) begun as early as 2011. MBSE Pathfinder and MBSE Infusion and Modernization Initiative (MAIAMI) are the examples of NASA efforts being taken for the last five years, just to name a few, while our project team believes that the APHMAL domain-specific language that is discussed on this project is another complimentary effort in this direction. The APHMAL could be a part of the PHM-for-HH&P solution with predictive screening capability, providing early and actionable real-time warnings on impending health issues that otherwise would have gone undetected unless a symptom is manifested or a sign is detected at a relatively late stage.

The project discusses the APHMAL definition, the metamodel of the domain-specific language, and transformation which is about customizing and mapping a standard sensor set to an individualized set for every crew member based on his/her pre-flight health screening results. The project also discusses language engineering and the transformation environment chosen.

2. Respective Contribution Made by the Project Team Members

(The section with a one-paragraph description of who did what for the project)

Breakdown of work by Project part:
Part 1 Source Code: Ian only
Part 2 Report: Section 1, Some of Section 4, Formatting & General editing: Alex. All other sections: Ian
Part 3 Video: Ian only
Part 4 Demo: Some slides prepared by Alex, technical descriptions and screenshots provided by Ian

ECSE-539 Advanced Software Engineering
Autonomous Prognostics and Health Management Automation Language (APHMAL)
Project (April 10, 2022)

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081
(the alphabetical order reflects also the contribution made by the team members in descending order)

3. Language Engineering and Transformation Environment

The section with a detailed two-page experience report about your chosen language engineering and transformation environment including a discussion on what went well and what did not go well.
The experience report will be written in the first person, from the experiential perspective of Ian Laffey

Note:
*For descriptions of Eclipse UI interactions **in bold**, note that this run on Mac OS with*
Eclipse Modeling Tools
Version: 2021-12 (4.22.0)
Build id: 20211202-1639
These UI interaction descriptions are added as helpful information, your UI experience may vary.

The first step towards creating the language engineering and transformation environment (henceforth referred to simply as "the environment") was to establish a working draft of the metamodel. This was done with eCore to begin with, as that is what I had the most experience using for metamodelling. After gathering feedback from Gunter, changes were implemented to the meta model and it was converted to Emfatic.

Emfatic is a part of Epsilon- the set of modeling languages and tools that we used to create our environment. Epsilon installation is done through the Eclipse installation manager. Installing the requisite packages was done by selecting **Help->Install New Software** on Eclipse, and entering the URL of the Epsilon Eclipse Install Site. Emfatic and ETL are a part of the Epsilon Core, and Flexmi was a separate installation, as tools in Epsilon are not included in the Epsilon Core. The only difference with installing Flexmi was a change of URL to the Flexmi Eclipse Install Site.

Now that Emfatic (and the rest of the Epsilon tools we will be using for our project) is installed through Eclipse, the features are available in Eclipse after a restart. The existing eCore model (.eCore extension) was easily mappable to Emfatic source (.emf extension) through **Eclipse -> Right Click on eCore Model -> Generate Emfatic Source**. Now that the Emfatic source had been generated, further changes could be made to the metamodel "on the fly" by editing the textual description of the metamodel. This was especially useful for the scope of our environment, because the qualityType enum could be easily changed by a user to add more qualityType options to the language specification.

In order to integrate Emfatic with other Epsilon based tools, it is necessary to use EPackages. Registering EPackages is done through Eclipse **Right Click on *.emf -> Register EPackages.** It is also worth noting at this point that EPackages use a URI, and specification of the package URI can be done through Emfatic. Registered EPackages are given the URI specified in their corresponding Emfatic source. Now that the EPackage has been generated, it can be referenced in all other Epsilon tools through a URI. The chosen URI for our EPackage was aphmal.

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081
(the alphabetical order reflects also the contribution made by the team members in descending order)

Now that our EPackage has been generated, our metamodel specification can be used with our other Epsilon based tools, languages, files. Any further changes to the metamodel can then be re-exported and picked up by the Epsilon based tools (A refresh may be necessary).

The next step is to create Input models using Flexmi (YAML flavor, indentation based). The YAML Flavor provides no functional/semantic difference from the standard Flexmi. It is merely to do with syntax of the terms. Defining a Flexmi file consists of specifying the URI of the metamodel (this links the model to the metamodel), and then specifying the model elements with the Flexmi YAML Flavor, Indentation Based syntax. More info about the syntax can be found at https://www.eclipse.org/epsilon/doc/flexmi/.

It is also worth noting here that Flexmi files do not currently support programmatic saving (resource.save()), and so I was unable to find a way to generate Flexmi models programatically. Flexmi is therefore used to create and specify Input models in a lightweight, user friendly (even for non-technical users), however Output models are saved as EMF .model files.

If a Flexmi model is created correctly, the Eclipse Outline View (Accessible through **Eclipse -> Window -> Show View -> Outline**) will display a view of the model (similar to a EMF .model file) with a tree-like containment view of model elements, references, attributes, etc. A couple of sample Flexmi models were created in order to test out Flexmi features and metamodel, some tweaks were made to the metamodel after seeing how actual model specification occurred, and more sample input models were planned for the future.

Now that we have established input models, the next task is the actual transformation itself. More details about the transformation will be discussed later in this report. In order to specify the transformation, Epsilon ETL was used (.etl file extension). ETL requires input and output models specified through **Right click on *.etl -> Run as -> Run configurations -> ETL Transformation -> Models tab -> Add.** Our input and output metamodel are the same (and referenced through apmhal EPackage), however the input file is specified as "*.flexmi", and the output filetype is specified as "*.model". In order to run ETL transformations, rules are necessary. More about rule syntax is available at: https://www.eclipse.org/epsilon/doc/etl/ It is worth noting that transformation use the form

```
rule name
transform s :input!Class
to t : output!Class{
(EOL Statements)+
}
```

where name can be anything, input refers to the name of the model for input (not the metamodel, the model name specified in ETL run configurations), output refers to the name of the output model, Class is a class in the metamodel. More about EOL Statements here: https://www.eclipse.org/epsilon/doc/eol/

Our model transformation necessitated an algorithm for mapping Sensors and Responders to Biomarkers, based on minimizing a sum of integer values in this mapped set (more about this elsewhere). ETL was not designed for such an algorithm, and I'm unfamiliar with EOL (Maybe EOL would have been a reasonable choice for implementing such an algorithm, but I am more comfortable in Java and so I used Java), so the algorithm was defined in Java.

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081
(the alphabetical order reflects also the contribution made by the team members in descending order)

Epsilon provides a way of integrating Java code into EOL statements. This is done through creating a plug-in project, adding an extension point to Epsilon tools, exporting the plug-in, adding the plug-in to Eclipse dropping, where it can then be called by the command
```
var sampleTool = new Native("org.eclipse.epsilon.examples.tools.SampleTool");
```
This creates sampleTool as a native Java object that we can use to pass in our input Biomarker list, and receive our output Device list. (Other things are passed to the method call to facilitate algorithm execution, but at a high-level, it just maps biomarker lists to device lists for a specific astronaut)
We execute this algorithm for each of our Astronauts as specified in our ETL Transformation. Note that static calls to native Java methods are also possible in ETL.

Now that this algorithm (which can be viewed as a black box by ETL), is completed the transformation is relatively simple. ETL syntax is very similar to ATL syntax. Running the ETL transformation produces an output model as expected. At this point, as debugging and testing of the transformation was occurring, additional sample input models were created for testing purposes.

Finally, a set of input models was chosen for submission (as opposed to testing) to demonstrate the features of our transformation.

What went well	What didn't go well
Able to create an input specification that is (I believe), very user friendly/ non technical	Setting up these tools is non-trivial. Need to look carefully through documentation to figure out how to set up the tools, and looking them up is of little use as they are not particularly mainstream currently.
Implement algorithm using rather complicated reflexive operations and deep inheritance structure. Learned a lot about eCore and Epsilon objects.	Typical errors due to coding while tired. Issues such as shallow vs deep copying of a list. Needing to create re-usable code (for sensor and responder lists). Actual sensor mapping algorithm wasn't completely trivial and some mistakes were made there.

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081
(the alphabetical order reflects also the contribution made by the team members in descending order)

Gained experience in working with complicated tools with little "google ability". Docs were pretty solid, but google did not work. Because of this I needed to really look through the documentation and understand what was going on as opposed to looking up individual components.	Transferring an on-paper algorithm (where you can say for example: "if sensor name is x") to an algorithm using EObjects, EDynamicObjectImpls, reflective method calls provides some challenges.
Was able to write transformation algorithm in Java, which was much easier for me than in another more domain specific language (such as ETL/ EOL). In addition, this transformation algorithm is highly modular. Someone else could define a better mapping algorithm in Java, and swap it out very easily while leaving the rest of our environment intact, allowing for efficiency based optimization.	Using the native Java calls in ETL transformations suppresses output (and I couldn't figure out how to get output to flush ever). Sometimes I had to resort to adding a null pointer exception within my Java plug-in in order to figure out if I had reached a conditional branch. In addition, needing to export the plug-in .jar, and add to dropins folder on Eclipse, then restart Eclipse added overhead to debugging process.
Going hands on with a custom, self made metamodel gave a new appreciation for the uses of certain meta-properties of eCore classes (particularly containment, uniqueness)	After "re-registering" an EPackage it is necessary to "refresh" EPackage references by going to all references, and deleting them and re-adding them. A feature in Epsilon for "refreshing" all references to exported EPackages of the same name would have been a great QOL change.
	All these strange behaviors and QOL issues that I documented had to be discovered on-the-fly and were not well documented. Some of them took many hours to discover through lots of trial-and-error.

4. Description of the Domain-specific Language Chosen : APHMAL

(The section with a two-paragraph description of your chosen domain-specific language.)

Alex is running the "PHM for HH&P" session on the annual IEEE Aerospace conference and suggested the domain-specific language for the project. The APHMAL could be a part of the PHM-for-HH&P solution with predictive screening capability, providing early and actionable real-time warnings on impending health issues that otherwise would have gone undetected.

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081
(the alphabetical order reflects also the contribution made by the team members in descending order)

The following are the assumptions made on the project in terms of current practice on the crewed space programs.

For the input/source/abstract syntax:

- Each crew member is recognized as a healthy person based on the formal pre-flight health screening routine and its results;
- Each crew member is provided with a set of his/her biomarkers;
- Biomarker is provided with a set of sensors, and a set of responders that “treat”/act upon the biomarkers. There must be at least 1 sensor and 1 responder for each biomarker.
- Devices must have qualityType specified of type == Astronaut.Quality in order to be selected to treat a biomarker. (1)

For the output/target/concrete syntax:

- The list of devices contained in an Astronaut all contain qualities of type Input!Astronaut.Quality
- The list of devices has been optimized to minimize the sum of qualities for the qualityType given by Input!Astronaut.Quality
- Each biomarker is covered by a responder and sensor (if (1) is met)

The purpose of the suggested and implemented transformation is to customize the standard set of devices for each crew member based on his/her pre-flight health screening results, so to optimize the sets in accordance with individualized health self-monitoring concept and the HH&P autonomy paradigm.

5. APHMAL Metamodel

(The section shows and discusses the metamodel of your domain-specific language. Include a diagram for the metamodel that is laid out nicely, so that it is readable. Note that the discussion must not repeat the information available in the metamodel itself, but rather cover the key design decisions taken when defining the metamodel.)
A .jpg diagram of the model has been included in our submission titled ‘eCSE539Project class diagram` However, given that we used Emfatic in order to specify the model, and our overall goals (outlined below), with the metamodel, I feel it makes more sense to include the textual specification as opposed to the diagram (it is very readable)

Key Design decisions/Overall goal of metamodel:

Since we will be performing a mapping to and from our DSL, it is important that our metamodel facilitate this. Also, to take advantage of Emfatic easy to read code, by using enums, users can easily add new

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081
(the alphabetical order reflects also the contribution made by the team members in descending order)

qualities to the DSL on the fly! The **overarching goal** of the meta-model was that it was simple, readable, and **extendable**.

```
@namespace(uri="apmhal", prefix="apmhal")
package eCSE539Project;

class APMHAL {
    val NamedElement[*] elements;
}

abstract interface NamedElement {
    !unique attr String name;
}

class Astronaut extends NamedElement {
    val Device[*] devices;
    ref Biomarker[*] biomarkers;
    attr QualityType[1] quality;
}

class Biomarker extends NamedElement {
    ref Sensor[+] sensors;
    ref Responder[+] responders;
}

abstract class Device extends NamedElement {
    ref Biomarker[*] biomarkers;
    val Quality[+] qualities;
}

class Responder extends Device {
//Further attributes/methods could go here
}

class Sensor extends Device {
//Further attributes/methods could go here
}

class Quality {
    attr int[1] value;
    attr QualityType[1] qualityType;
}

enum QualityType {
    weight = 0;
    height = 1;
    accuracy = 2;
}
```

Another important thing to keep in mind is that this simplification of the metamodel means lots of useful features could be added. We merely focused on the mapping of Devices to biomarkers,- why have separate Sensor and Responder classes, for example? The answer is that this metamodel, and environment in general is meant to be extendable/modular! Additional attributes and methods can be added differentiating Sensor and Responder, however in the context of our environment these were not necessary.

6. The Routine to Run the APHMAL Editor

(The section describes how to run your language editor, including an example model that shows all features of the language. In addition, list all source files you have worked on that are not fully generated. This includes the sample models – indicate which files are the sample models.)

Here is the routine to Analyze/Execute run the editor:

As discussed between Ian and Gunter, the editor merely consists of editing a Flexmi document, with all the features that Flexmi supports. As such the “editor” is Eclipse, with Epsilon and Flexmi installed, with the Epsilon view enabled, additionally with the **Outline** view enabled to ease with model creation.

SOURCE FILES AND DESCRIPTIONS IN README.MD

ALL INCLUDED SOURCE FILES ARE NOT FULLY GENERATED- WITH THE EXCEPTION OF *.MODEL & *.LAUNCH, MORE DETAILS ON IMPLEMENTATION IN VIDEO TUTORIAL

7. The Routine to Analyze/Execute APHMAL

(The section describes how to analyze/execute your chosen domain-specific language and lists all source files you have worked on that are not fully generated.)
Tutorial/Demo should provide additional context on this execution

As outlined in **Section 3.** the execution of APHMAL consists of:

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081
(the alphabetical order reflects also the contribution made by the team members in descending order)

1. Install Epsilon Core & Epsilon Flexmi (Optional step: Extend .emf metamodel to your liking)
2. Generate EPackage from Emfatic source
3. Define a model with Flexmi syntax
4. Specify this model as input model in ETL run config
5. Specify an output file path with same Emfatic metamodel in ETL run config
6. Run ETL transformation
7. Observe output file

Analysis of APHMAL files:

The metamodel can be generated as an eCore file by **Right click metamodel.emf -> Generate Ecore Model.** There is a direct mapping of Emfatic source to eCore model so the generated eCore model can be analyzed as a standard eCore model.

The ETL file is mostly just a wrapper for the Java method SampleTool object. A jar has been included (to be placed in drop-ins folder, restart required), and the source has also been included. This source file was handwritten by Ian Laffey in order to satisfy the problem of creating low cost device mappings for biomarkers. This could be re-written and re-compiled and "hot-swapped" into the model without changing anything else, as long as it takes the required inputs/output (needs to take a list of biomarkers, and return a list of sensors, these are represented as List<DynamicEObjectImpl>).

I attempted to write solid Java code, although I am not an expert on reflection, and eCore classes and Types are rather complicated. The code has numerous helper methods that assist in the main goal (creating a device mapping). The name SampleTool indicates that this is merely a Sample algorithm, it should be re-implemented before this entered a real life use case. I believe the Big O Time Complexity of the algorithm to be quite poor. I think it is exponential (due to the subset permutations), but I don't believe writing an efficient algorithm to be the purpose of this project.

8. The Languages Involved in the APHMAL Transformation

(The section with a two-paragraph description of each language involved in your transformation that is not discussed in the "Language definition" section.)

At the risk of writing superfluous information I will detail the languages used in the transformation, and what they do (at a high level).

Flexmi syntax is used to define the input model within the bounds of our metamodel. This can be viewed as the abstract syntax of our language. There are numerous constraints defined in the language definition. EOL Language is used in conjunction with ETL language for defining the transformation, which then calls our native Java algorithm. The output of the model can be viewed as a concrete syntax of our APMHAL, where sensors and responders are mapped to cover biomarkers.

ECSE-539 Advanced Software Engineering
Autonomous Prognostics and Health Management Automation Language (APHMAL)
Project (April 10, 2022)

Group #539P 2
Laffey, Ian ID# 260820791
Popov, Alex ID# 261057081
(the alphabetical order reflects also the contribution made by the team members in descending order)

9. Mapping from Source Language to Target Language

The table describes the mapping from source language to target language for your transformation.

The source language == target language. I think it makes more sense to view this transformation as an automated Concrete Syntax generation from Abstract Syntax definition.

Source: **Abstract Syntax**	**Target:** **Concrete Syntax**
APHMAL	DNE, merely used as a container for information necessary for Concrete Syntax
Astronaut	Astronaut
Device	Astronaut.Device (based on mapping algorithm)
Biomarker	DNE. Biomarker has been removed as it has been treated by responders and sensors
Device.Quality	Device.Quality (Qualities are preserved within the device)
Astronaut.Quality	DNE as all devices in Astronaut.Devices now contain that quality.