Extending PARMOREL to repair instances of Ecore models and some OCL constraints

Tore Berven[1,2]

[1] Western Norway University of Applied Sciences, Norway
[2] tore.berven@gmail.com

Abstract. In Model Driven Software Engineering (MDSE), models are created and used as part of the development process. The models contain implicit restrictions that define valid instances of the model. Models can also contain explicit restrictions, in the form of Object Constraint Language (OCL) constraints. Instance models that erroneously represents the domain, can be rectified by using specialized programs to perform model repair on the faulty instance model. PARMOREL is an extendable plugin for Eclipse, that utilizes machine learning techniques to find the best sequence of repairs for a model, based on user submitted preferences. PARMOREL currently supports the repairing of meta-models, but not on instances of models. This project attempts to extend PARMOREL to enable model repair on instances of models, as well as repairing violations of some OCL constraints, by first creating a plan for extending the framework and then attempting to implement it.

Keywords: PARMOREL · Model Repair · Model instances

1 Introduction

PARMOREL (Personalized and Automatic Repair of MOdels using REinforcement Learning) [6] is a plugin extension to the Eclipse Modelling Framework (EMF) [2] that utilizes machine learning techniques to repair Ecore models in accordance with the users preferred approach. The plugin receives a list of issues that have been detected in a model, along with a set of actions that can be performed to repair the given issues. PARMOREL then filters out which actions can be applied to fix each issue, and attempts to fix the model by applying sequences of actions to correct the issues, rewarding the actions that result in a model that is close to the user's specified preference. The sequence of repairing actions that achieve the highest scores are returned to EMF, which can then repair the model in accordance with the user's preferences.

PARMOREL currently supports repairing of Ecore meta-models, but not instances of the models. This project will attempt to extend PARMOREL to perform model repair on instances of models, as well as repairing violations of some OCL constraints.

2 Motivation

In Model Driven Software Engineering (MDSE), models are an important aspect that is used to visually represent the problem domain and documenting the system structure. Models that are modified over time will eventually start accumulating errors. Utilizing tools like PARMOREL to repair the broken models can simplify maintenance and ensures that the models are syntactically correct. Since models are used for documentation and, through the use of specialized software like EMF, to automatically generate code representing the system of the model, it becomes very important to ensure that the models are correct.

Instances of models can be used to represent examples of the modeled domain, helping developers and others to get a better understanding of the domain. Instances can also be used to document various program states, and it is important that the models are correct to prevent misconceptions from arising.

When creating instances of a model, there is nothing that prevents those instances from violating the implicit restrictions imposed by the meta-model. Invalid models could cause confusion, and possibly give an erroneous impression of the domain. Having tools that can automatically ensure the correctness of the instance models, could potentially prevent such mistakes from occurring. OCL constraints can also be used alongside a model, to define explicit restrictions on what is a valid instance.

Making PARMOREL able to process both meta-models and instances, would increase the overall usefulness of the plugin, and simplify the process for the user by only having one program/plugin to manage.

3 Methodology

To extend PARMOREL, we will make use of relevant literature [1, 4] written by former Master and PhD students that were responsible for the creation and development of PARMOREL, as well as the source code for the PARMOREL plugin found in GitHub repository [8].

To keep development efforts separate from the primary plugin, we will clone the code to a new GitHub repository [9]. The new repository will, however, only contain the source code for PARMOREL, and needs to be combined with the plugin code found at GitHub repository [7] in order to be usable. The primary reason for keeping the plugin code and the source code separate, is that we avoid including unnecessary elements in the repository that are not directly related to the extension project. Another reason, is that it becomes easy to exchange the PARMOREL source code with a different version, keeping things modularized.

(Must elaborate on how they should be combined in later section!)

We will start with a thorough examinations of the source code to get an overview of how the various components are connected, and to get an understanding of how PARMOREL operates when repairing a model. Using the information gained in combination with the provided literature, we will attempt to form a plan on how to modify PARMOREL to enable repair of instance models.

After having formed the plan, we will attempt to implement it, and if successful, test the extended PARMOREL by repairing a simple instance model which violates some implicit and OCL constraints. As this project is meant to be a proof of concept rather than a full implementation, we will limit ourselves to one test model.

4 Design and Implementation

PARMOREL is divided into three main modules [1]: the modelling module, the preferences module, and the learning module. The modeling module, which is responsible for processing the input models and sending information about the model to the learning module, is further divided into an issues module and an actions module. The learning module, responsible for deciding which actions to apply to the input model and storing experience for later use, is also divided into an experience module and an algorithm module.

According to chapter 3.9 in [4], to extend PARMOREL to enable repair of additional model types, we need to make changes to the following components: Model, ModelType, ErrorExtractor, ActionExtractor and ModelProcessor.

- Model abstracts the representation of the input model.
- ModelType is an enum that specifies the type of the model, and also keeps track of error codes that are not supported for that model type.
- ErrorExtractor is responsible for extracting any errors from the input model, and communicate any errors found to the learning module.
- ActionExtractor is responsible for getting the actions that can be applied to the input model, to fix the detected errors.
- ModelProcessor is responsible for applying actions to the input model, and initializing the Q-table with actions that can fix the errors.

These components abstract the input model for the rest of the plugin, and by implementing them with support for instance models, we should be able to enable PARMOREL to repair instances of meta-models.

4.1 The plan

We start by creating a new class *EcoreInstanceModel* to represent the instance model. Because an ecore instance model is derived from an ecore meta-model, and the only difference between an *EcoreInstanceModel* and an *EcoreModel* is the model type, we implement *EcoreInstanceModel* as a subclass of *EcoreModel*. We then need to change the access modifiers of the variables *model* and *modelCopy* in *EcoreModel*, from private to protected to make them accessible.

We continue by adding a new enum "ECORE_INSTANCE" to *ModelType*, which is initialized with an empty set of unsupported error codes. In the future, if any error codes are known to be unsupported by PARMOREL for repair on instance models, the error code id's can be added to the set to make it

explicit that the error type is not supported. Afterwards, we can override the getModelType() method in *EcoreInstanceModel* to return ECORE_INSTANCE.

To get errors from the input model, we will make use of EMF diagnostican [3], which is used in EMF to validate models. Since *EcoreErrorExtractor* already makes use of EMF diagnostician to get errors from the model, we make a subclass *EcoreInstanceErrorExtractor* which initializes the *unsupportedErrorCodes* variable using the ECORE_INSTANCE model type. We then need to change the access modifier of the *unsupportedErrorCodes* variable in *EcoreErrorExtractor* from private to protected to make it accessible for the subclass.

ModelProcessor Because instance models are represented differently compared to meta-models, we need to modify the ModelProcessor component to be able to process the input instance models. We make a copy of *EcoreModelProcessor* which we name *EcoreInstanceModelProcessor*, and make it a subclass of *EcoreModelProcessor*. In the subclass constructor, we call the parent constructor with the provided knowledge, before initializing the *errorExtractor* field variable to a new *EcoreInstanceErrorExtractor*. We then change the access modifiers of the variables *knowledge*, *errors* and *errorExtractor* in *EcoreModelProcessor* from private to protected, to make them accessible to the subclass.

We then override and modify the initializeQTableForErrorsInModel() method, to check if the given model is an instance of *EcoreInstanceModel*, and if so, we cast the model to an *EcoreInstanceModel* and pass it on (initializeQTableForErrorsInModel((EcoreInstanceModel) model)).

We move on to private method initializeQTableForErrorsInModel(EcoreModel model), where we change the input parameter type from *EcoreModel* to *EcoreInstanceModel*. In the first if statement in the following for-loop, we change the *ModelType* from which the method fetches unsupportedErrorCodes, from ECORE to ECORE_INSTANCE.

Then we override the tryApplyAction() method, and modify it to check if the given model is an instance of *EcoreInstanceModel*.

In the private tryApplyAction() method, we need to cast the model context objects to something else instead of EPackage, since instance model context objects are implementations. However, we did not manage to find the correct class to cast the instance model context objects to before the deadline of the project.

For the remainder of *EcoreInstanceModelProcessor*, we would need to remove most of the methods related to meta-model modifications, and add new methods for applying actions to instance models.

QModelFixer We then need to notify *QModelFixer* to use the classes we have made. To do so, we make a new class *EcoreInstanceQModelFixer* and make it a subclass of *EcoreQModelFixer*. We then change the access modifiers of the field variables *uri* and *resourceSet* in *EcoreQModelFixer* from private to protected, to make them accessible to *EcoreInstanceQModelFixer*.

The constructors of *EcoreInstanceQModelFixer* should be changed to support modifying of instance model files [10], by changing "resourceSet.getResourceFactoryRegistry() .getExtensionToFactoryMap().put("ecore", new EcoreResourceFactoryImpl());" to "resourceSet.getResourceFactoryRegistry() .getExtensionToFactoryMap().put("xmi", new XMIResourceFactoryImpl());".

Methods initializeModelFromFile() and getModel(File model) should be changed to return new *EcoreInstanceModel*. Similarily, updateRewardCalculator() and initializeModelProcessor() should return new *EcoreInstanceModelProcessor*; initializeErrorExtractor() should return a new *EcoreInstanceErrorExtractor*.

To make everything more organised, we gather the new classes in a new package: no/hvl/projectparmorel/qlearning/ecore/instances

4.2 Repairing OCL violations

To get the actions that can be applied to the input model to fix the errors, PARMOREL attempts to apply all available actions that can modify the model, on the offending context elements, and stores the actions that manages to fix the error. PARMOREL already manages to find actions to fix most errors, but because OCL constraint violations are resulting from the breaking of invariants specified by model developers, we have to determine repairing actions based on the violation.

We create a new class *EcoreInstanceActionExtractor* and make it a subclass of *EcoreActionExtractor*. We make the constructor in *EcoreInstanceActionExtractor* call the parent constructor, and override the extractActionsNotInQTableFor() method, to allow for the processing of OCL constraint violation errors, before passing the remaining errors to *EcoreActionExtractor* and merging the lists of actions.

EMF diagnostician reports broken OCL invariants as errors with error code 0. We can therefore sort out the OCL constraint violations by looping over the error list, and process all errors with code 0.

To make PARMOREL utilize the modified ActionExtractor, we need to replace all occurrences of *EcoreActionExtractor* with *EcoreInstanceActionExtractor* in *EcoreInstanceModelProcessor* and *EcoreInstanceQModelFixer*.

5 Evaluation

To test the extended PARMOREL, we need to combine it with the parts that compose the remainder of the plugin. We first clone the contents of GitHub repository [7], and then replace the PARMOREL module with our extension version.

To enable the repairing option for instance models, we need to open plugin.xml, and change the value on line 45 in the image below from 'value= "ecore"', to 'value= "*"'.

This makes the repair alternative visible for all file elements, which was necessary because model instances have different file endings, and we were unable to

```
<extension
      point="org.eclipse.ui.menus">
   <menuContribution
         locationURI="popup:org.eclipse.ui.popup.any">
      <menu
            icon="icons/parmorel.png"
            id="hvl.plugin.parmorel.menues.main"
            label="Parmorel">
         <command
               commandId="hvl.plugin.parmorel.commands.repairCommand"
               icon="icons/bandage.png"
               label="Repair"
               style="push">
         </command>
         <visibleWhen
               checkEnabled="false">
            <iterate
                  ifEmpty="false"
                  operator="or">
               <test
                     property="org.eclipse.core.resources.extension"
                     value="ecore">
               </test>
            </iterate>
         </visibleWhen>
      </menu>
   </menuContribution>
```

Fig. 1. An excerpt from the file plugin.xml

find a common factor that could be used to identify instance models. A negative side effect of making the repair option visible to all file elements, is that trying to initiate repairs on a file that is not a model will cause PARMOREL to hang, making it unresponsive to future repair commands. Future work should find a way to prevent this from happening.

The test model is a simple representation of a bowling game, with one implicit violation and four OCL constraint violations. The OCL constraints were integrated with the meta-model using OCLinEcore [5].

The meta-model specifies that each matchup must consist of two games. The instance model however, will not prevent the user from adding additional games, making the model invalid. With OCLinEcore we have added additional constraints that limits:

- Player names must be less than 30 characters.
- Player height must be above 0 and below 400.
- Player date of birth must not be null.

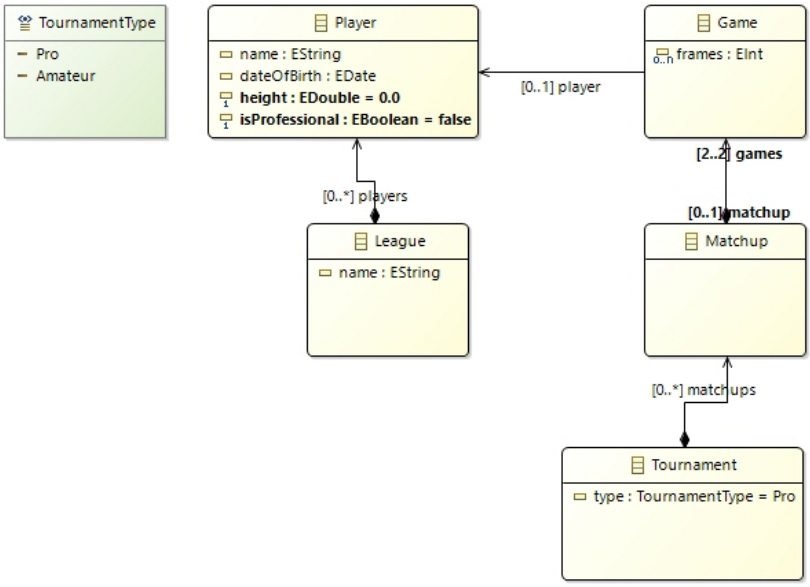

Fig. 2. A diagram of the test model

```
import ecore : 'http://www.eclipse.org/emf/2002/Ecore';

package bowling : _'org.eclipse.modelrepair.bowling' = 'https://org/eclipse/modelrepair/bowling'
{
    class Player
    {
        attribute name : String[?];
        attribute dateOfBirth : ecore::EDate[?];
        attribute height : ecore::EDouble[1];
        attribute isProfessional : Boolean[1];
        invariant NameShouldNotBeTooLong: name.size() < 30;
        invariant PlayersCannotBeTooTall: height < 400.0;
        invariant PlayersMustHaveAValidHeight: height > 0.0;
        invariant PlayersMustHaveABirthDate: dateOfBirth <> null;
    }
```

Fig. 3. The OCLinEcore invariants

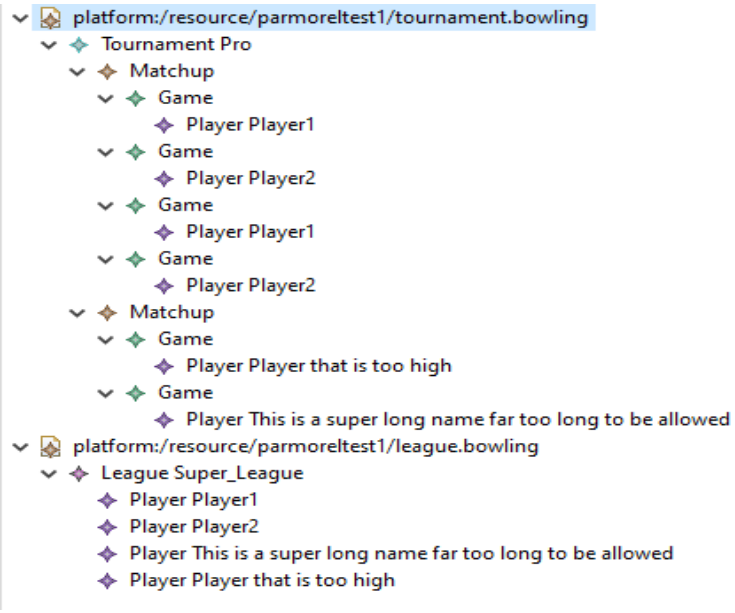

Fig. 4. The broken instance models

6 Discussion

Because of the time constraints of the project, we were from the offset limited to adding support for only a few OCL constraint violations; but in the end, we were unable to get a functioning proof of concept before the project deadline. The remainder of this discussion will therefore focus on findings made during testing and development of the plugin extension.

EMF diagnostician was able to detect the OCL constraint violations when validating the league.bowling instance model, but was only able to detect the implicit violation when validating tournament.bowling. This may be due to tournament.bowling using league.bowling as a loaded resource to include the player references, prompting EMF diagnostican to ignore the players, as they were not considered part of the instance model.

Surprisingly, EMF diagnostican was able to detect violations of the OCL constraints, in addition to the implicit errors. This might be due to the way the OCL constraint invariants were annotated as part of the meta-model using OCLinEcore.

Having the invariants be part of the model simplified the repairing process, by not making PARMOREL dependent on the input of an additional file containing the OCL constraints. As a consequence, PARMOREL may not be able to detect OCL constraint violations in cases where the OCL invariants are stored outside the meta-model. We leave it to future work to confirm whether or not this is the

case, and to further develop PARMOREL to support repair of OCL constraint violations on instance models.

7 Conclusion

We have attempted to create a plan for extending PARMOREL to repair Ecore instance models, but did not manage to complete the plan or make a functioning proof of concept before the project deadline.

PARMOREL is well modularized, making it easy to get a good overview of the code, though a few of the classes are quite big and difficult to understand. EcoreModelProcessor was quite challenging to process when attempting to extend PARMOREL, and figuring out how to apply actions to instance model components turned out to be perhaps the most difficult part of the project.

We hope that the results from this project can help others continue our work with extending PARMOREL to repair instance models.

The project code from attempting to implement the plan, can be found in the referenced GitHub repository [9].

References

1. Barriga A., Heldal R., Rutle A., Iovino L. (2021) *PARMOREL: A framework for customizable model repair*. Submitted to Journal of Software and Systems Modeling (SoSyM).
2. Eclipse Modeling Framework (EMF), `https://www.eclipse.org/modeling/emf/`
3. EMF diagnostician java docs, `https://download.eclipse.org/modeling/emf/emf/javadoc/2.6.0/org/eclipse/emf/ecore/util/Diagnostician.html`
4. Marthinsen, M. (2020) *Automatic Model Repair Using Machine Learning*. Master thesis. Bergen: Western Norway University of Applied Sciences/University of Bergen. Available from: `https://ict.hvl.no/wp-content/uploads/2020/09/Masteroppgave_Magnus_Marthinsen.pdf`
5. OCLinEcore wiki, `https://wiki.eclipse.org/OCL/OCLinEcore`
6. PARMOREL main page, `https://ict.hvl.no/project-parmorel/`
7. Repository for PARMOREL plugin, `https://github.com/MagMar94/ParmorelEclipsePlugin`
8. Repository for PARMOREL source code, `https://github.com/MagMar94/projectparmorel/tree/f1f01a94ec1dc99cd75e645591ac3752ea33c9b3`
9. Repository for proof of concept (project code), `https://github.com/Tore422/DAT355\_Project\_ExtendingPARMOREL`
10. StackOverflow, modifying instance models from code, `https://stackoverflow.com/questions/30034826/create-and-modify-ecore-files-and-their-instances-from-code`