Towards Round-Trip Engineering of Code Fragments Embedded in Models

Sultan Almutairi
Department of Computer Science, University of York
York, North Yorkshire, United Kingdom
soha500@york.ac.uk

Athanasios Zolotas
School of Computer Science and Mathematics, Liverpool John Moores University
Liverpool, Merseyside, United Kingdom
a.zolotas@ljmu.ac.uk

Dimitris Kolovos
Department of Computer Science, University of York
York, North Yorkshire, United Kingdom
dimitris.kolovos@york.ac.uk

ABSTRACT

While embedding code fragments in abstract software models (e.g. Java code in UML models) is far from ideal, it remains a commonly-employed approach for achieving full model-based code generation. In this paper, we embrace this reality and present an approach for extending model-to-text (M2T) transformation languages with support for round-trip engineering of such code fragments. The approach consists of a new construct in M2T templates named *sync regions*, and a mechanism for synchronising hand-written code in sync regions with the source model of the M2T transformation. We have implemented the proposed approach on top of an existing M2T language (Epsilon Generation Language) and we have carried out experimental evaluation of the correctness and performance of our implementations. The obtained results suggest that the synchronisation algorithm scales linearly with the number of sync regions.

KEYWORDS

Model-Driven Engineering, Model-to-Text Transformation, Round-trip Engineering

ACM Reference Format:

Sultan Almutairi, Athanasios Zolotas, and Dimitris Kolovos. 2022. Towards Round-Trip Engineering of Code Fragments Embedded in Models. In *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3550356.3561578

1 INTRODUCTION

Model-to-text (M2T) transformation is routinely used in model-based software engineering processes to generate implementation-level artefacts such as executable code[1], documentation and configuration scripts from abstract, typically domain-specific, models in an automated and repeatable manner.

A common way to implement M2T transformations is using dedicated *template-based* languages such as Acceleo [2], Java Emitter Templates [1], Xpand [9], Velocity [6] and StringTemplate [13]. Similarly to server-side scripting languages such as PHP and ASP.NET, M2T languages provide first-class support for combining static content with text computed from the elements of one or more input models, and can offer improved readability compared to imperative M2T transformation programs implemented using string concatenation [14].

Often, modelling languages do not provide sufficient expressive power to capture all the information required to achieve full code generation. In such cases, developers are called to choose among one of the following options:

- Complement the generated code with hand-written code that adds the missing information, using *protected regions*, inheritance or delegation;
- Extend the abstract and concrete syntax of the modelling language with concepts required to capture the missing information within the model, ideally at an implementation-agnostic level of abstraction;
- Minimally extend the modelling language to allow modellers to embed code fragments written in the target implementation language within their models (e.g. embed C++ code within UML models [18]).

While the latter approach of embedding *uninterpreted strings*, as Stephen Mellor calls them in [10], in models is detrimental in terms of model analysability and portability, it remains a popular approach among practitioners [18] due to its low upfront implementation cost, the desire to maintain the model as the *single source of truth*, familiarity with the target implementation language, and aversion to complicating the syntax of the modelling language. Even the widely-used Stateflow modelling environment uses embedded C and MATLAB strings to implement control logic in state charts[2].

When this road is chosen, the next dilemma that developers come to face is in which environment to write the code embedded in their models. Writing the code within the modelling environment deprives them of essential features such as code completion and error reporting. On the flip side, writing the code within an IDE incurs an additional overhead of having to copy and paste it back to the modelling tool, and also involves a risk that they forget to do so and code fragments are accidentally overwritten next time the M2T transformation is executed. As a result, the latter, which involves a manual synchronisation process, can lead to inconsistencies and mistakes as it is admittedly a tedious and error-prone task.

[1]In this paper we use the terms *M2T transformation* and *code generation* interchangeably

[2]https://www.mathworks.com/help/stateflow/ref/chart.html

MODELS '22 Companion, October 23–28, 2022, Montreal, QC, Canada
© 2022 Association for Computing Machinery.
This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in *ACM/IEEE 25th International Conference on Model Driven Engineering Languages and Systems (MODELS '22 Companion), October 23–28, 2022, Montreal, QC, Canada*, https://doi.org/10.1145/3550356.3561578.

In this paper, we present an approach that enables developers to use target-language-aware IDEs to write such fragments by automating the synchronisation process between generated-then-manually-extended source code files, and the models from which they were generated. The proposed approach is agnostic of both the modelling language and the target implementation language. A proof-of-concept prototype has been developed on top of an existing template-based M2T language, the Epsilon Generation Language (EGL) [15]. More specifically, our proposed approach takes as input the models used in the M2T transformation and the files generated from the M2T templates, and propagates the hand-written code added to the latter back to the model, automating the process of round-trip synchronisation.

The remainder of the paper is organised as follows. In Section 2, we provide an example of M2T transformation to concretely illustrate the problem targeted by this research. In Section 3, we present the proposed approach and its implementation, explaining how changes made to generated files are synchronised with the source models of the M2T transformation. Section 4 reports on the applicability and limitations of the approach. In Section 5 we evaluate the correctness and performance/scalability of the proposed approach. In Section 6 we present related work, and Section 7 concludes and outlines plans for future work.

2 MOTIVATING EXAMPLE

To concretely demonstrate the problem we are targeting in this work and to motivate the proposed approach, we present an example involving a minimal M2T transformation, where round-trip synchronisation between the source model and the generated source code is desirable.

For this example, we use a minimal component-connector domain-specific language (DSL), the abstract syntax of which is illustrated in Figure 1. In our DSL a system consists of components and connectors. Each component has many input ports (*inPorts*) and one output port (*outPort*). Each port has a name and a type, and ports can communicate through connectors. Each connector has exactly one source port and one target port.

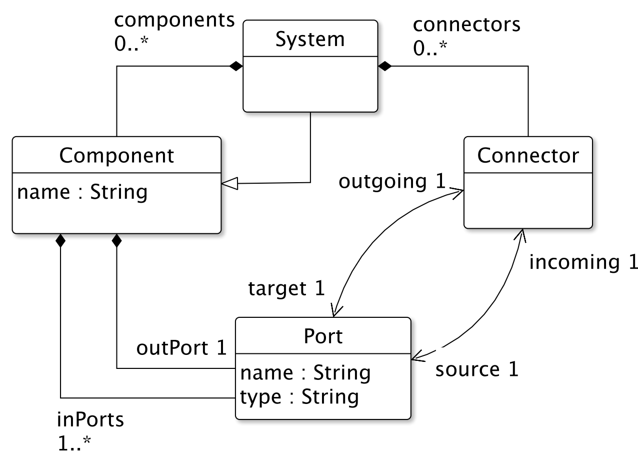

Figure 1: Metamodel of the Component-Connector DSL

```
1  rule System2Class
2      transform s : System {
3      template : "../common/system2class.egl"
4      target : "src-gen-sync-regions/syncregions/" + s.name
              + ".java"
5  }
6
7  rule Component2Class
8      transform c : Component {
9      template : "sync-regions-component2class.egl"
10     target : "src-gen-sync-regions/syncregions/" + c.name
              + ".java"
11 }
```

Listing 1: EGL rules for generating Java code from component-connector models

Figure 2 shows a model that conforms to our DSL and which captures a small part of the operation of a water heating boiler. The system (model) has two components: *TemperatureController* and *BoilerActuator*. Also, it has three input ports (namely, *temperature*, *targetTemperature*, and *boilerStatus*). The *TemperatureController* component receives input from two ports (*temperature* and *targetTemperature*). It computes the difference between the two and the result is propagated to the *BoilerActuator* component along with the current status of the boiler. The *BoilerActuator* component decides whether to turn the boiler on or off.

From models like the one shown in Figure 2, we wish to generate executable Java code. We achieve this through a template-based M2T transformation, implemented using EGL, and shown in Listings 1-3. Although we use EGL in this example, the transformation could be implemented using any other template-based M2T language. The program in Listing 1 consists of two rules. The first rule, in lines 1-5 is used to generate a Java class for every model element of type *System*. Line 1 gives the rule a name, line 2 contains the name of the type, instances of which the rule should transform, line 3 declares the template that will be used for the transformation, and line 4 specifies where the generated file will be stored. The second rule, in lines 7-11 is used to generate one Java class for each component in the system.

The template invoked by the *System2Class* rule is shown in Listing 2. Line 1 prints the class name. Lines 2-8 generate an *execute()* method that has one parameter for each input port of the system and returns a value, the type of which is the same as the type of the output port of the system. The list of the input parameters for each component is calculated using a utility operation *getInputParameters()* defined in lines 12-24. The second template is for the *Component2Class* rule and is shown in Listing 3. Line 1 prints the class name and lines 2-4 generate an *execute()* method for the component with appropriate input parameters and return type, and an empty body. When we execute the transformation on the model of Figure 2, it produces the files shown in Listings 4 and 5 for the system, and the *BoilerActuator* component, respectively[3].

[3]A very similar class is generated for the *TemperatureController* component, which we omit to reduce unnecessary repetition.

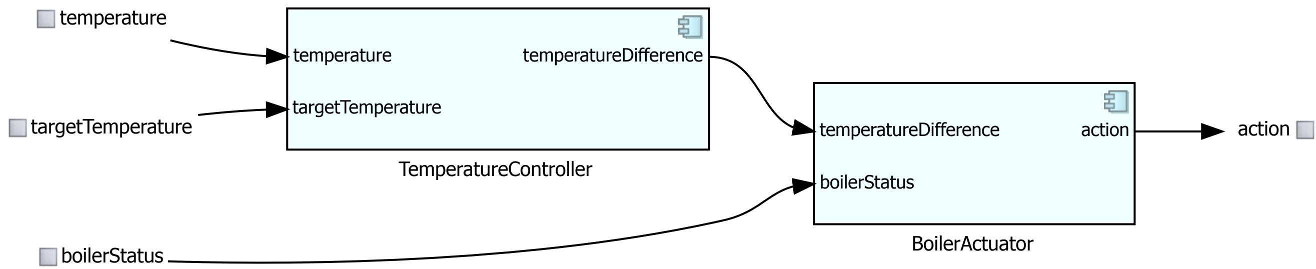

Figure 2: BoilerController model that conforms to the metamodel in Figure 1

```
1  public class [%=s.name%] {
2    public [%=s.outPort.type%] execute([%=s.inPorts.
         collect(p|p.type + " " + p.name).concat(", ")%])
          {
3    [%for (child in s.components){%]
4    [%=child.name%] [%=child.name.ftlc()%] = new [%=child.
         name%]();
5    [%=child.outPort.type%] [%=child.name.ftlc()%]Result =
          [%=child.name.ftlc()%].execute([%=child.
         getInputParameters().concat(", ")%]);
6    [%}%]
7
8    return [%=s.outPort.incoming.source.eContainer().name.
         ftlc() + "Result"%];
9    }
10 }
11 [%
12 operation Component getInputParameters(){ {
13   var parameters : Sequence;
14   for (p in self.inPorts) {
15     if (p.incoming.source.eContainer().isTypeOf(Model))
          {
16       parameters.add(p.incoming.source.name);
17     }
18     else {
19       parameters.add(p.incoming.source.eContainer().
              name.ftlc() + "Result");
20     }
21   }
22   return parameters;
23 }
24 %]
```

Listing 2: EGL template that generates a Java class realising the communication between components of the system

While the model contains sufficient information[4] to generate the content of the *execute()* method of the *BoilerController* as per Listing 4, it has no means of expressing the behaviour of each individual component. Hence, the generated *execute()* method of the *BoilerActuator* class in Listing 5 is empty.

[4]With many assumptions e.g. regarding ordering and freedom from cycles which are necessary to keep this example minimal.

```
1  public class [%=c.name%] {
2    public [%=c.outPort.type%] execute([%=c.inPorts.
         collect(p|p.type + " " + p.name).concat(", ")%]) {

3
4    }
5  }
```

Listing 3: EGL template for generating a Java class for each individual component

```
1  public class BoilerController {
2    public int execute(int temperature, int
         targetTemperature, boolean boilerStatus) {
3    TemperatureController temperatureController = new
         TemperatureController();
4    int temperatureControllerResult =
         temperatureController.execute(temperature,
         targetTemperature);
5    BoilerActuator boilerActuator = new BoilerActuator();
6    int boilerActuatorResult = boilerActuator.execute(
         temperatureControllerResult, boilerStatus);
7
8    return boilerActuatorResult;
9    }
10 }
```

Listing 4: Generated class for BoilerController component

```
1  public class BoilerActuator {
2    public int execute(int temperatureDifference, boolean
         boilerStatus) {
3
4    }
5  }
```

Listing 5: Generated class for BoilerActuator component

2.1 Implementing Component Behaviour

To add the missing behaviour, one could extend the generated code with hand-written code using inheritance or delegation, or even directly edit the generated files using *protected regions* (discussed in Section 6). This approach would leave some information about the system out (i.e. the behaviour of individual components) of the model, which would no longer be the system's *single source of truth*. Keeping the model as the single source of truth of the system can be desirable, for example:

- to enable full code generation as part of a headless continuous integration (CI) process,
- to avoid storing generated code in the version control repository,
- to simplify change review and authorisation (it is easier for team members to review changes if they are centralized in the model and are not dispersed across the model and code files)

In addition, although using inheritance might be a potential solution to the problem, it is not the case that all languages are supporting such a concept (i.e., HTML does not offer such a mechanism). Thus, this approach *limits the target languages that a M2T transformation engine can support*.

An alternative approach would be to extend our component-connector language so that models conforming to it can specify the behaviour of individual components. As discussed in the previous section there are two ways to achieve this. The first one is to extend the language with appropriate syntax constructs for capturing complex behaviour (e.g. add *IfStatement*, *ArithmeticExpression*, *VariableDelcaration* concepts to its metamodel), effectively turning it into a programming language. The second – which, as discussed in the previous section, practitioners commonly opt for – is to extend the *Component* type, with a *behaviour* string attribute, as shown in Figure 3, that can be used to specify the behaviour of components directly in Java and which will be emitted as-is by the M2T transformation in the body of the component's *execute()* function.

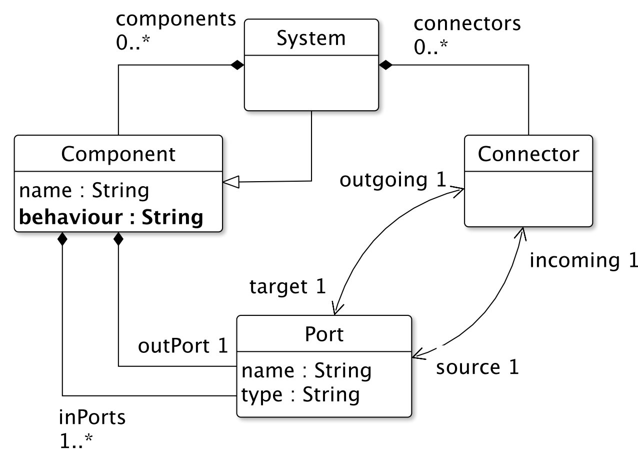

Figure 3: Extended version of the metamodel of Figure 1 with a new *behaviour* attribute

Assuming that a developer opts for the latter, they are now faced with the dilemma of which environment to use to write these embedded Java code fragments that implement the behaviour of individual components. One option is to write this code within the modelling tool, effectively forfeiting code completion, and error reporting. The other option is to generate code from a version of the model where the behaviour attributes of the components are empty, then define the behaviour of each component in a Java-aware IDE, and once the behaviour has been satisfactorily implemented and tested using all the tools offered by the IDE, copy the hand-written code back into the behaviour attributes of the respective components. In this case, the template of Listing 3, would need to be extended with an additional EGL statement in line 3, which would emit the value of the new *behaviour* attribute to the generated code as-is.

Beyond compromising model analysability and portability (since the model is now bound to a specific target language) – which can be acceptable compromises in many cases – every time the developer makes changes to the *execute()* function of a component within their Java IDE, they need to remember to copy and paste the updated code back into the *behaviour* attribute of the respective component in the model. This is a tedious and error prone task that we set out to automate in this work.

3 EXTENDING M2T TEMPLATES WITH SYNC REGIONS

To automate synchronisation between generated-then-edited files and their source models in scenarios such as the one discussed in the previous section, we propose extending M2T templates with *sync regions*. A sync region is a region in a generated file which is appropriately fenced using identifiable start/end comments, and encloses content that needs to be kept in sync with a specific slot (pair of model element and attribute) in the model. The key difference between a *sync region* and a *protected region* is that modified text within a protected region is preserved by retaining it in the generated file upon re-generation, while modified text within a sync region is preserved by automatically copying it *into* the model.

In this section we discuss how we have extended EGL with support for sync regions, however, the same principles can be used to extend any other template-based M2T language in a similar manner. Our prototypical extension works with EMF-based models [16] persisted in the XMI format, where each element has a unique persistent ID, and is limited to M2T transformations that consume a single model as input.

3.1 Specifying Sync Regions

We have extended EGL with two additional methods to specify sync regions:

- *startSync(String startComment, String id, String attribute)*: Emits a single-line comment in the target file, starting with the *startComment* character sequence (e.g., // for Java), which denotes the start of a sync region, and contains the *id* of the model element and the name of its *attribute* that the content of the sync region needs to be kept in sync with (e.g., *behaviour*, for the example presented in Section 2). A variant of the method with an extra *endComment* parameter

```
1 public class [%=c.name%] {
2   public [%=c.outPort.type%] execute([%=c.inPorts.
        collect(p|p.type + " " + p.name).concat(", ")%]) {

3   [%=out.startSync("//", c.id, "behaviour")%]
4   [%=c.behaviour%]
5   [%=out.endSync()%]
6   }
7 }
```

Listing 6: Extended version of the template of Listing 3 with a sync region

```
1 public class BoilerActuator {
2   public int execute(int temperatureDifference, boolean
        boilerStatus) {
3   //sync _bfpnFUbFEeqXnfGWlV2_8A, behaviour
4
5   //endSync
6   }
7 }
```

Listing 7: The result of executing the template of Listing 6 against the *BoilerActuator* component

is also available to accommodate languages that require both a prefix and a suffix for their comments (e.g., “<!– comment –>" for HTML).

- *endSync()*: Emits a single-line comment that marks the end of the active sync region.

The use of the methods above is demonstrated in Listing 6, which is an extended version of our original Listing 3 template that generates Java classes from individual components. In the extended version of the template, three new lines have been added (lines 3-5) and the output of executing it against the *BoilerActuator* component is shown in Listing 7. Line 3 of the template produces the comment in line 3 of the generated file, which denotes the start of a sync region. The generated comment starts with the // character sequence as instructed by the first argument of the *startSync* method. It continues with the *sync* keyword which allows the synchronisation engine (which is described in Section **??**) to distinguish sync region comments from general comments in the generated file, followed by the element identification token (i.e., the ID of the component) [5]. Finally, the name of the attribute against which the content of the sync region should be synchronised (e.g., *behaviour*) is given. Line 4 prints the content of the *behaviour* attribute of the component (empty in the initial version of our model), and Line 5 produces the *//endSync* comment in the generated file, that denotes the end of the sync region.

[5]For example, the (*_bfpnFUbFEeqXnfGWlV2_8A* ID is an auto-generated XMI ID produced by the Eclipse Modelling Framework, that was used to implement the component-connector DSL and the sample instance model.

```
1  public class BoilerActuator {
2    public int execute(int temperatureDifference, boolean
         boilerStatus) {
3    //sync _bfpnGUbFEeqXnfGWlV2_8A, behaviour
4      if (temperatureDifference > 0 && boilerStatus ==
           true) {
5        return 1; // turn boiler off
6      }
7      else if (temperatureDifference < 0 && boilerStatus
           == false) {
8        return 2; // turn boiler on
9      }
10     else return 0; // do nothing
11   //endSync
12   }
13 }
```

Listing 8: Extended *BoilerActuator* class with hand-written behaviour

3.2 Synchronising Sync Regions with Model Elements

A developer can now specify the behaviour of the *BoilerActuator* component within the produced sync region of the generated *BoilerActuator* Java class, as shown in lines 4-10 of Listing 8, benefiting from modern IDE features such as code completion and syntax highlighting.[6]

Once they have made the desirable changes to the behaviour of generated components, the next step is to trigger a synchronisation mechanism (the second part of our proposed approach), which will identify and copy the hand-crafted behaviour into the *behaviour* attributes of the respective components in the source model. A core requirement of the proposed approach is that between generation and synchronisation, the source model is not edited in any way. We describe the synchronisation algorithm below.

3.3 Synchronisation Algorithm

The synchronisation alorithm consists of three steps which help in locating sync regions, validating their consistency and finally updating the model. These three steps are described below.

Step 1: Sync Region Identification. The synchronisation algorithm receives three inputs:

- the root directory under which files generated by the M2T exist;
- character sequences that denote the start of a comment (e.g. //);
- the EMF model to be synchronised with any manual updates made to sync regions of the generated files.

The algorithm recursively scans all files under the root directory and identifies sync regions that start and end with appropriate comments. For each sync region, the algorithm checks that:

[6]We have used numbers instead of named constants in lines 5, 8 ans 11 to keep the example concise.

```
1  <h1>BoilerActuator</h1>
2   ...
3  <pre class="code">
4  <!--sync _bfpnGUbFEeqXnfGWlV2_8A, behaviour-->
5   if (temperatureDifference > 0 && boilerStatus == true)
        {
6     return 1; // turn boiler off
7   }
8   else if (temperatureDifference < 0 && boilerStatus ==
        false ) {
9     return 2; // turn boiler on
10  }
11  else return 0; // do nothing
12 <!--endsync-->
13 </pre>
```

Listing 9: Generated *BoilerActuator* HTML documentation file with component behaviour

- the element ID and attribute of the region are specified and correspond to valid elements and mutable attributes in the model;
- the text within the sync region can be converted to a value compatible with the type of the respective attribute.

If any of the regions are found to not meet the criteria above, the algorithm exits with an appropriate error message and without updating the EMF model.

Step 2: Sync Region Consistency Checking. In principle, the same attribute of the same model element can appear in multiple sync regions across the generated code-base. This can be the use due to either a user error or because the contents of the same model attribute is used in more than one places in the generated text. Thus, before we update the model, we need to ensure the consistency of sync regions that refer to the same element and attribute. If they are found to:

- have the same value, they are marked as consistent.
- have two distinct values and one of them is the same as the value of the attribute in the source model, then the other (different) value is marked as the "new" value for the attribute.
- two distinct values, none of which corresponds to the value of the attribute in the model, or more than two distinct values, they are marked as inconsistent.

To demonstrate a concrete case where such an inconsistency may be encountered, consider that we extend the Java-generating M2T transformation discussed above, to also generate a HTML file for each component of the system, which presents the component in a graphical form (e.g. using a Mermaid JS diagram that visualizes the component, its input and output ports and other components connected directly to it) and a section that contains a copy of the Java behaviour of the component, also within a sync region. A fragment of the generated BoilerActuator.html file would look like Listing 9.

If a developer now modifies the content of the two sync regions for the behaviour of BoilerActuator in Listings 8 (e.g. to improve the structure of the code) and 9 (e.g. to fix a typo in a comment) to two different values, none of which matches the value of the behaviour attribute in the model, the synchronisation algorithm is not able to decide whether to update the behaviour of the BoilerActuator in the model to the new content found in the Java file or in the HTML file.

Upon the detection of an inconsistency, the synchronisation algorithm exits with an appropriate error message that includes information on the file that the error was found in and the conflicting values in the file and model. Thus, no updates are made to the model. It is expected that the user will remedy these inconsistencies manually for the synchronisation algorithm to run to completion again.

Step 3: Model Updating. At this point, sync regions have been verified to be well-formed (step 1) and free of conflicts (step 2). As such, the algorithm can proceed with updating the attributes of the model elements to which sync regions refer. For each element/attribute involved, the content of the respective sync region is coerced to the type of the attribute and the coerced value is assigned to the attribute. Once all elements/attributes have been updated, the model is saved to disk.

4 APPLICABILITY AND LIMITATIONS

We now discuss the applicability of the proposed approach and its main limitations.

Model Element Identities. As discussed in Section **??**, the proposed approach requires model elements to have unique, persistent, and immutable identities, as these are used to trace sync regions back to model elements of interest. In our experience, the majority of modelling tool support such identities (e.g. XMI-IDs in EMF, GUIDs in PTC Integrity Modeller). However, we are also aware of tools, such as Matlab Simulink, where exposed model element IDs are path-based and can change when elements are moved/renamed in a model, and where – as a consequence – the proposed technique is not applicable.

Metamodel and Model Pollution. Sync regions need to be backed by respective attributes in the metamodel. As such, the more sync regions are introduced, the more the metamodel, and the models that conform to it, will be polluted with implementation-level information. With reference to our running example, as long as all the behaviour of a component can fit within the body of the *execute()* method, extending the metamodel with a *behaviour* attribute feels like a reasonable compromise. However, if changes need to be made to other parts of the component class too (e.g. new import statements, fields, utility methods), then the metamodel and the M2T transformation need to be extended with respective attributes and sync regions for each such part, which can feel increasingly uncomfortable. In our view, this is an inherent issue of this approach (i.e. not limited to this particular example) and needs to be taken into consideration before its adoption. For M2T transformations that require the generated code to be augmented in several places, other integration techniques such as inheritance/delegation may be more appropriate.

Metamodel Evolution. The generated comments that mark the start and end of sync regions use the name of the attribute with which the content in the region must be synced. If the metamodel evolves and the attribute is renamed, retyped in a breaking way (e.g. from String to Integer), or disappears altogether, the reference implementation will report an error and it will be up to the developer of the M2T transformation to rectify any inconsistent sync regions in previously generated files.

Embedded Code Consistency. Since code fragments embedded in sync regions are copied verbatim between sync regions and the model, changes made to the model can invalidate the embedded code fragments, making them uncompilable, or even worse, inadvertently changing their semantics. For example, if the *temperatureDifference* port of the *BoilerActuator* component of Figure 2 is renamed to *tempDiff* in the model, when the code is re-generated, the body of the execute method will produce compilation errors as it will still refer to the temperature difference variable with its old name.

Direct Changes to the Embedded Code Fragments. It is important to highlight that in the proposed approach the code fragments must not be directly modified by users in the model as in this case the synchronisation algorithm will treat the content of sync regions as the "new" values and will overwrite any directly modified code fragments in the model the next time it is executed.

Deleted Sync Regions. While the synchronisation algorithm can cope with inconsistently updated and malformed sync region markers, it cannot cope with sync regions being deleted altogether from generated files. Ideally such missing sync regions should be reported to the user but this cannot be achieved without keeping additional metadata outside the generated files, which is undesirable.

5 EVALUATION

This section outlines the results of our evaluation[7] of the correctness, generalisability, performance and scalability of our prototype implementation of the proposed approach and reflects on its applicability and known limitations.

5.1 Correctness

We have argued about the completeness of the proposed approach analytically in the previous sections by describing how it copes in different scenarios (e.g. malformed or inconsistent sync regions) and by listing its known limitations. To build confidence on the correctness of our prototype too (i.e. the fidelity with which it implements the presented approach), we have developed several unit tests using the JUnit library to ensure that the synchronisation algorithm behaves as expected under normal circumstances (well-formed and conflict-free sync regions) and gracefully fails when models or generated files are modified manually in inconsistent ways.

Table 1 demonstrates the result of our tests. Each test scenario presented in Table 1 was duplicated to test for all the types (e.g., String, integer, double, float, and Boolean). The proposed approach was able to complete the synchronisation process successfully and as expected when the sync regions were well-defined and when there were no conflicts in the values included in the generated files. In addition, the algorithm has successfully identified all the inconsistencies that were introduced and reported the errors back to the user, as expected.

In the second part of the correctness test was focusing on the *syntax of the sync regions*. The possible cases that could happen in this aspect can be found in Table 2. Table 2 also demonstrates the result of our tests. The results show that the proposed approach can act when any of the possible cases happened and warn the developers where the mistake is by reporting a clear message into the console with information on where the error is found and the conflicting values involved.

5.2 Performance and Scalability

To assess the performance and scalability of our prototype implementation, and to ensure that it is free from unnecessary bottlenecks, we executed the M2T transformation discussed in Section 2 on models of various sizes, producing 5 sets of files ranging from 2,000 to 10,000 files (with a step of 2,000 files).

Each experiment was repeated 3 times, collecting the average time and memory, for each of the following scenarios [8]:

- Each generated file had *one* sync region.
- Each generated file had *two* sync regions.
- Each generated file had *three* sync regions.
- Each generated file had *four* sync regions.
- Each generated file had *five* sync regions.

In this experiment, the values included in the generated files were *always different from those stored in models*, thus our approach had to update the values in every element in the source model. We have chosen to evaluate our solution with up to 10,000 files, a number that significantly exceeds the number of files produced by typical M2T transformations in our experience. The results are summarised in Table 3 and in Figures 4-7.

Figure 4 presents the results of the average total time required for the execution of the synchronisation for the 5 different sizes of generated file sets and the 5 scenarios with the different number of sync regions. Each line represents one file set, while the horizontal axis is the number of sync region(s) for each file in the set. As one can see, the execution time increases linearly as the number of sync regions increase for all the different sets of files. However, it is not clear if the execution time increases linearly or has a exponential trend when the number of files increases while keeping the number of sync regions the same (see Figure 6). This is an indication that there might be a glitch in the implementation of the proposed approach when dealing with big number of files and needs to further be investigated. However, having 10,000 generated files is rarely the case even in extreme M2T transformation cases.

In terms of absolute values, in the scenario of having 2,000 files with 1 sync region in each of them, our approach required about 40 seconds to complete the synchronisation (see Table 3 - top highlighted value). For the largest of the experiments (i.e., having 10,000

[7] All unit tests, input models, generated files, raw and analysed results for all the experiments presented in this section are available at https://github.com/soha500/EglSyncNew/

[8] The experiments were executed on a laptop computer with the following specifications: MacOS Mojave 106.14., Intel Core i7, 2-cores @ 3.5Ghz, 1x16 GB 2133MHz LPDDR3 RAM

Test ID	**# Sync Regions (SR)**	**Description**	**Expected Output**	**Pass**
T1	1	Model and SR have the same value	Model value is not updated	Pass
T2	1	Model and SR have different values	Model is updated with the SR value	Pass
T3	2	Model and both SRs have the same value	Model value is not updated	Pass
T4	2	Both SRs have the same value but different from the Model	Model is updated with the SRs value	Pass
T5	2	One SR has the same value with the Model but the second SR has different value	Model is updated with the second's SR value	Pass
T6	2	SRs have different values and both are different with the value in the Model	Inconsistency Error	Pass
T7	3	Model and all SRs have the same value	Model value is not updated	Pass
T8	3	All SRs have the same value but different from the Model	Model is updated with the SRs value	Pass
T9	3	One SR has the same value with the Model, the other two have the same value but different from the Model	Model is updated with the values of the two SRs that have the different valeus	Pass
T10	3	All SRs have different values	Inconsistency Error	Pass

Table 1: The results of testing the possible cases when using sync regions (inconsistent values).

Tests	**Syntax of sync region**	**Expected Output**	**Pass**
i	ID element is missing in one sync regions	At least on sync region does not contain ID	Pass
ii	Attribute is missing in one sync regions	At least on sync region does not contain attribute	Pass
iii	Beginning of at least one sync region is missing	An error syntax	Pass
iv	End of at least one sync region is missing	An error syntax	Pass
v	Respective element is missing	The respictive element not found	Pass
vi	Respective attribute is missing	The respective attribute is not found	Pass
vii	Different type	Incompatible type	Pass

Table 2: The results of testing the syntax of sync regions (misformated or incompleted).

# Files	**# Sync Regions**	**Average (Total) Time (in s)**	**Average Memory Used (in MB)**
2000	1	**40.83**	316.77
2000	2	88.61	58.97
2000	3	124.60	48.43
2000	4	168.39	283.53
2000	5	212.60	276.61
4000	1	167.89	97.24
4000	2	345.54	75.94
4000	3	545.45	94.69
4000	4	740.26	852.79
4000	5	950.55	546.58
6000	1	384.27	**1131.14**
6000	2	754.12	968.73
6000	3	1192.69	599.44
6000	4	1736.60	1030.33
6000	5	2224.22	955.47
8000	1	724.76	968.26
8000	2	1510.34	827.48
8000	3	2341.29	842.78
8000	4	3188.45	777.47
8000	5	4221.06	680.05
10000	1	1198.81	244.07
10000	2	2368.72	596.18
10000	3	3678.04	624.46
10000	4	5000.96	507.28
10000	5	**6366.22**	231.46

Table 3: Average execution time and memory consumption form the different number of files and sync regions

files with 5 sync regions each) the average time taken for the 3 runs was around 1 hour and 45 minutes (6,366.22 seconds). Even in this extreme scenario the synchronisation completes successfully.

Figure 6 presents the same data but this time as the number of generated files is increasing for each of the scenarios for the same number of sync regions. Each line in Figure 6 represents a scenario with fixed number of sync regions and the 5 data points on the line represent the 5 different sizes of file sets.

Finally, in Figures 5 and 7 the average memory consumption is presented as the number of sync regions increase (keeping the number of files fixed) and as the number of files increases (keeping the number of sync regions fixed), respectively. There is no clear correlation, which is explained by the fact that Java garbage collection is clearing up memory when needed. What is of importance, is that in the worst case our prototype solution consumed about 1GB of memory (1131.14MB - see highlighted value in the memory consumption column of Table 3).

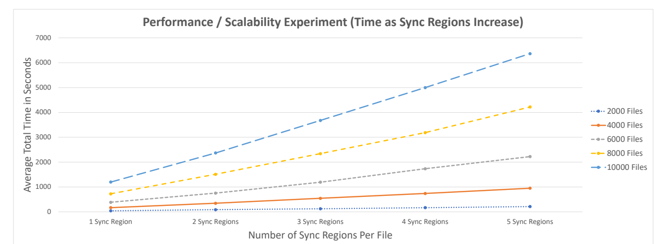

Figure 4: Results of measuring the average time for different size of models and number of sync regions as the number of sync regions increases.

5.3 Generalisability

To assess the generalisability of the synchronisation algorithm, we adapted the M2T transformation to generate files for different languages such as Java, Python, HTML, and Ruby. We tested the synchronisation prototype solution by repeating the synchronisation

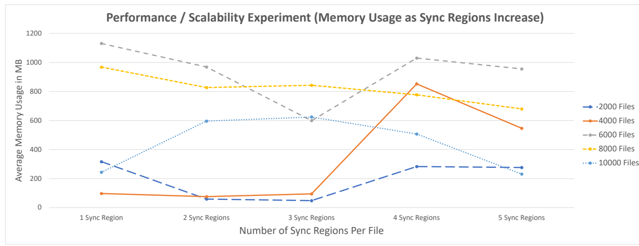

Figure 5: Results of measuring the average memory for different size of models and number of sync regions as the number of sync regions increases.

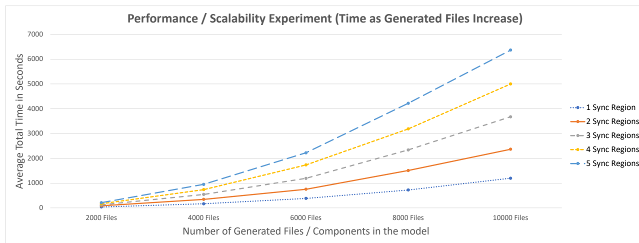

Figure 6: Results of measuring the average time for different size of models and number of sync regions as the number of files increases.

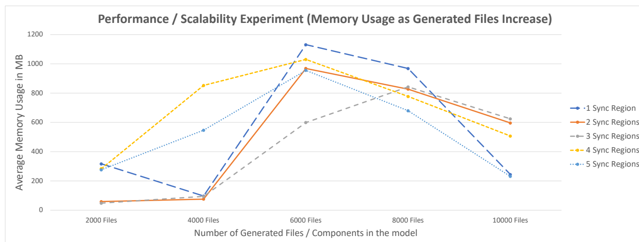

Figure 7: Results of measuring the average memory for different size of models and number of sync regions as the number of files increases.

Target language	Opening Comment Format	Closing Comment Format	Test Result
Java	// or /*	*/	Pass
HTML	<!–	–>	Pass
Python	#	N/A	Pass
Ruby	# or =begin	=end	Pass

Table 4: Generalisability experiment results

(in the form of JUnit tests) 100 times for each of the aforementioned programming languages. In this experiment, the content of *all* regions in the generated files was different to the corresponding values in the source model. Table 4 presents the open/close comment format used for each of the target programming languages and the results (i.e., the synchronisation algorithm passed all the tests and updated the models as expected).

5.4 Threats to Validity

The results obtained through the experiments described in this section are consistent with our expectations based on the inherent (linear) complexity of the proposed synchronisation algorithm when sync regions increase. However, it is not clear if the same linear trend exists in the case where the number of files increases. This requires further investigation to identify if there is any bottleneck in the implementation. To build additional confidence on the generalisability of the results, the prototype would benefit from evaluation in more extreme scenarios (e.g. with a large number of sync regions per file, with large individual sync regions).

5.5 User Evaluation

Beyond evaluating the correctness, performance and scalability of the proposed *sync regions* approach, it would also be useful to verify that it delivers the expected benefits to modellers in practice. This would require an experimental setup that would involve developing a domain-specific language and a supporting generator, training users in their use and then measuring whether the benefits of being able to edit code fragments within a target-language-aware IDE outweigh the burden of the extra step to run the synchronisation process described in Section **??**. As the benefits of using a modern IDE compared to a plain text editor are well-recognised, we argue that the value of such an experiment is minimal.

Comparing the effectiveness of sync regions with other approaches such as protected regions or inheritance/delegation-based generated code augmentation would be much more interesting but also very challenging to generalise as essentially the benefits of sync regions would boil down to the value of maintaining the model as the single source of truth of the system. Arguably, this can vary significantly depending on the extent to which code fragments need to be embedded in models, the size and complexity of such fragments and the programming language in which they are written, as well as the team's established processes and domain-specific constraints (e.g. certification requirements). To extract any generalisable observations, a series of such experiments involving different DSLs, generators and domains would need to be carried out, which is beyond the scope of this paper.

6 RELATED WORK

Greifenberg et al. [7] summarised known mechanisms for integrating generated and handwritten code for object-oriented programming languages:

- Generation gap is a pattern that keeps the handwritten and generated code separate by putting them in different classes using inheritance [3] or delegation [4]. More specifically, delegation is a pattern of object composition in object-oriented programming (OOP) [4]. It consists of two objects: a delegator and a delegate. The delegator works by delegating parts of its functionality to the delegate by invoking the latter's methods. To achieve this, the delegate provides an interface that declares the method signatures that can be invoked. The *include* mechanism that languages such as PHP (but not Java) provide is another approach for integrating handwritten and generated code. However, not all the languages support such mechanisms (e.g., HTML, LATEX) and as a

result such approaches cannot be applied for all the target languages.

- Some object-oriented languages (such as C#) provide support for partial classes, i.e. classes split across multiple source code files (some of which can be generated and some of which can be hand-written) that are combined into a single logical class when the application is compiled.
- A part-merger is a syntax-aware component that is able to merge multiple files of a certain type such as Java files, into one file [8, 19].
- Protected regions are regions that are declared in the generation templates by the developers for the purpose of adding handwritten code to the generated files and protecting it upon regeneration [15]. Many M2T transformation languages provide support for protected regions such as EGL, XPand [17] and Acceleo [2]. Gascuena et al. [5] mentioned the importance of having a mechanism that would enable developers to manually add handwritten code and presented protected areas as a solution.

The sync regions approach presented in this paper is most similar to protected regions as in both approaches generated and handwritten content co-exists in the same files. The main difference is that while in protected regions hand-written content is preserved by the M2T transformation engine upon regeneration, in sync regions it is automatically copied back into the model in a separate synchronisation step, thus restoring the model as the single source of truth.

Action and executable (modelling) languages, like the Foundational Subset for Executable UML Models (fUML) [12] and the Action Language for Foundational UML (Alf) [11] can eliminate the need for injecting general-purpose language code fragments in models, improve model analysability and facilitate multi-target code generation. On the flip side, the editors/IDEs of such languages tend to provide inferior support for features such as code navigation, completion and debugging compared to the respective facilities of general-purpose programming languages. Also, this requires developers to learn yet another language – often with limited documentation/examples compared to general-purpose programming languages.

7 CONCLUSION AND FUTURE WORK

In this paper, we have presented an approach that facilitates automated synchronisation between models and textual artefacts generated from them via template-based M2T transformation. We have implemented the proposed approach on top of an existing M2T language (EGL) and we have conducted preliminary evaluation of its scalability. The results of our evaluation experiment suggest that the synchronisation algorithm scales linearly with the number of sync regions but further experimentation is required to identify if increasing the number of files has an exponential impact in the execution time; a primary investigation has already been carried out while a full investigation is included in our plans for future work. Going forward, we plan to extend the proposed approach so that it can detect changes that have been made by developers outside of sync regions in generated files, so that they are not inadvertently overwritten the next time the M2T transformation is executed. User evaluation, in the context described in Section 5.5, would also be of interest.

ACKNOWLEDGMENTS

The work in this paper has been partially funded through the HI-CLASS InnovateUK project (contract no. 113213).

REFERENCES

[1] Eclipse Foundation. 2011. *"Java emitter templates" [Online].* https://projects.eclipse.org/projects/modeling.m2t.jet
[2] Eclipse Foundation. 2022. *Acceleo [Online].* www.eclipse.org/acceleo
[3] Martin Fowler. 2010. *Domain-specific languages.* Pearson Education.
[4] Erich Gamma. 1995. *Design patterns: elements of reusable object-oriented software.* Pearson Education.
[5] Jose M Gascuena, Elena Navarro, Patricia Fernández-Sotos, Antonio Fernandez-Caballero, and Juan Pavon. 2015. IDK and ICARO to develop multi-agent systems in support of Ambient Intelligence. *Journal of Intelligent & Fuzzy Systems* 28, 1, 3–15.
[6] Joseph D Gradecki and Jim Cole. 2003. Mastering Apache Velocity.
[7] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiss, Alexander Roth, et al. 2015. A comparison of mechanisms for integrating handwritten and generated code for object-oriented programming languages. In *2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD).* IEEE, 74–85.
[8] Timo Greifenberg, Katrin Hölldobler, Carsten Kolassa, Markus Look, Pedram Mir Seyed Nazari, Klaus Müller, Antonio Navarro Perez, Dimitri Plotnikov, Dirk Reiß, Alexander Roth, et al. 2015. Integration of handwritten and generated object-oriented code. In *International Conference on Model-Driven Engineering and Software Development.* Springer, 112–132.
[9] Benjamin Klatt. 2007. Xpand: A closer look at the model2text transformation language. , 2008 pages.
[10] Stephen J. Mellor, Stephen Tockey, Rodolphe Arthaud, and Philippe Leblanc. 1999. An Action Language for UML: Proposal for a Precise Execution Semantics. In *The Unified Modeling Language. «UML»'98: Beyond the Notation*, Jean Bézivin and Pierre-Alain Muller (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 307–318.
[11] Object Management Group. 2017. Action Language for Foundational UML (Alf). OMG Standard.
[12] Object Management Group. 2021. Semantics of a Foundational Subset for Executable UML Models (fUML). OMG Standard.
[13] T. Parr. 2013. *"StringTemplate," [Online].* https://www.stringtemplate.org/
[14] Louis M Rose, Nicholas Matragkas, Dimitrios S Kolovos, and Richard F Paige. 2012. A feature model for model-to-text transformation languages. In *Proceedings of the 4th International Workshop on Modeling in Software Engineering.* IEEE Press, 57–63.
[15] Louis M Rose, Richard F Paige, Dimitrios S Kolovos, and Fiona AC Polack. 2008. The epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications.* Springer, 1–16.
[16] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. 2009. *EMF: Eclipse Modeling Framework 2.0* (2nd ed.). Addison-Wesley Professional.
[17] Eugene Syriani, Lechanceux Luhunu, and Houari Sahraoui. 2018. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures* 52 (2018), 43–62.
[18] Johannes Trageser. 2020. On the Need for a Formally Complete and Standardized Language Mapping between C++ and UML. In *Proceedings of the 15th International Conference on Evaluation of Novel Approaches to Software Engineering - ENASE,.* INSTICC, SciTePress, 540–547. https://doi.org/10.5220/0009578305400547
[19] Steffen Zschaler and Awais Rashid. 2011. Towards Modular Code Generators Using Symmetric Language-Aware Aspects. In *Proceedings of the 1st International Workshop on Free Composition* (Lancaster, United Kingdom) *(FREECO '11).* Association for Computing Machinery, New York, NY, USA, Article 6, 5 pages. https://doi.org/10.1145/2068776.2068782