Modularity Mechanisms in Flexmi

1 Introduction

To create and edit models conforming to object-oriented metamodels defined using languages such as MOF and Ecore, users need to be provided with supporting tools that offer appropriate concrete (e.g. diagrammatic, textual, table/tree-based) syntaxes. Such tools can be generic and language-independent, or they can be bound to a specific metamodel. In the EMF ecosystem, metamodel-independent tools include the built-in reflective tree-based editor, an implementation of the Human-Usable Textual Notation (HUTN) [1, 2], and EMF's built-in support for the XML Metadata Interchange (XMI) syntax [3]. In terms of language-specific tools, textual syntaxes can be defined using frameworks such as Xtext [4] and MontiCore [5], diagrammatic syntaxes can be supported using frameworks such as Sirius, Graphiti and GMF, and form-based editors can be constructed using EMF Forms or EMF Parsley [6].

The main advantage of metamodel-specific tools is the ability to precisely tailor the desired concrete syntax(es). However, they require an upfront investment to develop, and continuous effort to maintain and keep up to date with their underpinning frameworks (e.g. Xtext, Sirius). Metamodel-independent concrete syntaxes on the other hand require no tool development and maintenance effort but can be more rigid and verbose.

Arguably, different approaches are more suitable depending on the expertise and preferences of the intended users, the maturity, size and nature of the metamodel in question, and the available resources to develop and maintain bespoke tooling.

In this line of work we are interested in metamodel-independent textual concrete syntaxes. In this niche, the available options in the EMF ecosystem are currently XMI and the OMG's Human Usable Textual Notation. In [7] we introduced a new metamodel-independent XML-based textual syntax called Flexmi, which is supported by a fuzzy parser that does not require exact name correspondence between the names of types and features in the metamodel and the tag and attribute names used in the XML representation of a model. In this paper, we report on enhancements to Flexmi with respect to modularity and reuse.

The rest of the paper is organised as follows. Section 2 provides an overview of the Flexmi textual concrete syntax and Section 3 discusses extensions of Flexmi that aim at enhancing modularity and reuse. Section 4 concludes the paper.

2 An Overview of Flexmi

In this section we provide an overview of Flexmi as introduced in [7], using the Simulink-like component-connector *comps* metamodel presented in Figure 1 as

a running example, and a minimal model of a speed monitoring component that conforms to the said metamodel, which is graphically illustrated in Figure 2.

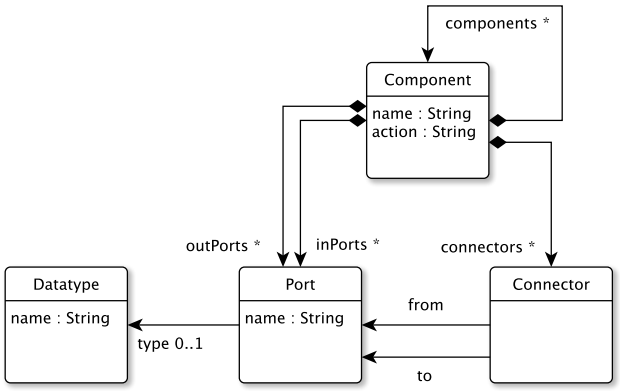

Fig. 1. The *comps* metamodel

The component in Figure 2 has two input ports (speed and location) and an output port (warning). The incoming (geo-)location is forwarded to a *SpeedLimitCalculator* component which returns the speed limit for the location. The speed is then compared against the limit using the nested *Comparator* component and if it exceeds it, a warning is produced.

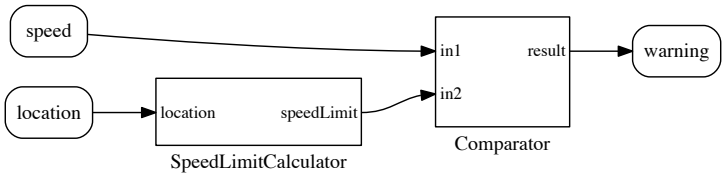

Fig. 2. A speed monitoring component that conforms to the *comps* metamodel

Listing 1.1 illustrates a Flexmi representation of the model in Figure 2. As discussed in [7], line 1 declares that the model is an instance of the *comps* metamodel. When the Flexmi parser encounters the *comp* tag in line 2, it compares it against the names of all classes in the metamodel, it finds that the closest match is *Component* and therefore creates an instance of that type. In line 3, the parser compares the *in* tag against the names of all containment references of *Component* and all the valid contained types and finds that the best match is the *inPorts* reference, hence it creates an instance of *Port* (which is the only valid type for this reference) and adds it to the values of the reference. The same

tolerant fuzzy matching approach is used to map XML attributes to attributes and non-containment references. For example, the *n* attribute in line 2 is mapped to the *name* attribute of Component while the *f* and *t* attributes in lines 18-22 are mapped to the *from* and *to* non-containment references of class *Connector*. The complete parsing algorithm that Flexmi uses is discussed in detail in [7].

```
1  <?nsuri comps?>
2  <comp n="SpeedMonitor" >
3    <in n="speed"/>
4    <in n="location"/>
5    <out n="warning"/>
6
7    <comp n="SpeedLimitCalculator">
8      <in n="location"/>
9      <out n="speedLimit"/>
10   </comp>
11
12   <comp n="Comparator" a="result = in1 > in2">
13     <in n="in1"/>
14     <in n="in2"/>
15     <out n="result"/>
16   </comp>
17
18   <con f="SpeedMonitor.location"
19     t="SpeedMonitor.SpeedLimitCalculator.location"/>
20   <con f="speed" t="in1"/>
21   <con f="speedLimit" t="in2"/>
22   <con f="result" t="warning"/>
23 </comp>
```

Listing 1.1. Speed monitor model in Flexmi

3 Adding Modularity to Flexmi

To capture larger models, the ability to split a model over multiple files is often desirable. To address this need, Flexmi has been extended with two new XML processing instructions to enable importing and including content from other EMF models.

3.1 Importing Models

Suppose that we wish to specify the types of the ports of the speed monitoring component in Listing 1.1. To render the type definitions reusable, we opt to record them in a separate Flexmi file called *types.flexmi* (the name of the file has no special semantics), which appears in Listing 1.2. The underscore tag in line 2 is a placeholder to accommodate models (such as this one) which have more than one root elements (very similarly to XMI's *xmi* root tag).

```
1  <?nsuri comps?>
2  <_>
3    <type n="boolean"/>
4    <type n="geo"/>
5    <type n="float"/>
6  </_>
```

Listing 1.2. types.flexmi

We can now import *types.flexmi* using an *import* processing instruction (line 2) from the speed monitoring model file and refer to its elements by id as illustrated in Listing 1.3.

```
1  <?nsuri comps?>
2  <?import types.flexmi?>
3  <comp n="SpeedMonitor" >
4    <in n="speed" t="float"/>
5    <in n="location" t="geo"/>
6    <out n="warning" t="boolean"/>
7    ...
8  </comp>
```

Listing 1.3. Speed monitor model with port types in Flexmi

At this point, it is worth contrasting Flexmi's file-level model imports to XMI's model-element level references. While in Flexmi, *types.flexmi* needs to be imported once, in the equivalent XMI presented in Listing 1.4, the path of the imported file needs to be repeated every time a reference to one of its elements is made (lines 3, 6 and 9). To the best of our knowledge, HUTN provides no syntax for file-level imports either.

```
1  <comps:Component xmlns:comps="comps" name="SpeedMonitor">
2    <inPorts name="speed">
3      <type href="types.xmi#/2"/>
4    </inPorts>
5    <inPorts name="location">
6      <type href="types.xmi#/1"/>
7    </inPorts>
8    <outPorts name="warning">
9      <type href="types.xmi#/0"/>
10   </outPorts>
11   ...
12 </comps:Component>
```

Listing 1.4. Speed monitor model with port types in XMI

3.2 Including Models

We now wish to specify the internal structure of the *SpeedLimitCalculator* component (see Figure 2 and lines 7-10 of Listing 1.1). The intended inner structure of the component is shown in Figure 3.

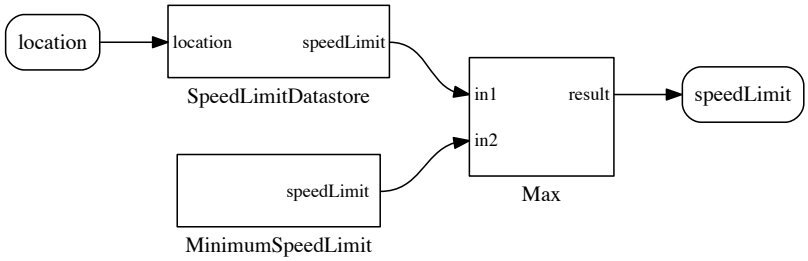

Fig. 3. The inner structure of the *SpeedLimitCalculator* component

If we wish to do this in a separate model file (e.g. *slc.flexmi* as shown in Listing 1.5), we can use Flexmi's *include* processing instruction – which operates in a very similar way to LaTeX's *input* command – which instructs the Flexmi parser to parse the contents of *slc.flexmi* as if they were a part of the main model file. That is, we can replace lines 7-10 of Listing 1.1 with an `<?include slc.flexmi?>` processing instruction.

```
1  <?nsuri comps?>
2  <?import library.flexmi?>
3  <comp n="SpeedLimitCalculator">
4    <in n="location"/>
5    <out n="speedLimit"/>
6
7    <comp n="MinimumSpeedLimit">
8      <out n="speedLimit"/>
9    </comp>
10
11   <comp n="SpeedLimitDatastore">
12     <in n="location"/>
13     <out n="speedLimit"/>
14   </comp>
15
16   <comp n="Max" a="result = (in1 > in2) : in1 ? in2">
17     <in n="in1"/>
18     <in n="in2"/>
19     <out n="result"/>
20   </comp>
21
22   <!-- connections omitted -->
23
24 </comp>
```

Listing 1.5. SpeedLimitCaclculator in Flexmi (slc.flexmi)

3.3 Reusable Model Element Templates

The *Comparator* and *Max* components in Figures 2 and 3 are very similar in terms of their structure as both of them contain two input ports called *in1* and *in2* and an output port called *result*. Given that the *comps* language does not define a notion of component inheritance, the only option available with a format such as XMI or HUTN is to tolerate this near-duplication.

To reduce duplication in such cases, without needing to extend the instantiated metamodel, Flexmi introduces a templating mechanism. In Flexmi, templates are specified using the reserved `<\_template>` XML tag. A template can have a number of named parameters and a content nested tag under which its content is defined. With reference to our example, we create a new Flexmi file (*library.flexmi*) which contains the definition of a *binary_ operator* template.

```
1  <?nsuri comps?>
2  <_>
3    <_template name="binary_operator">
4      <parameter name="in"/>
5      <parameter name="out"/>
6      <content>
7        <comp>
8          <in name="in1" type="${in}"/>
9          <in name="in2" type="${in}"/>
10         <out name="result" type="${out}"/>
11       </comp>
12     </content>
13   </_template>
14 </_>
```

Listing 1.6. The binary-operator template (library.flexmi)

The template has two parameters (*in* and *out* in lines 4 and 5) and its content is a component (line 7) with two input ports (line 8 and 9) and one output port (line 10). The types of these ports are controlled by the value of the *in* and *out* parameters using string replacement. Templates can be instantiated in models by name using an underscore prefix. Attributes of template invocations that also start with an underscore are passed as values for their named parameters, while all other attributes override the respective attributes of the template content element. In the context of our running example, we can replace lines 12-16 of Listing 1.1 with the following line (after importing *library.flexmi*):

```
1  <_binary_operator n="Comparator" a="result = in1 > in2" _in="
       float" _out="boolean"/>
```

Listing 1.7. The *Comparator* component of Listing 1.1 replaced with an instantiation of the *binary_ operator* template

and lines 16-19 of Listing 1.5 with the following line:

```
1  <_binary_operator n="Max" a="result = (in1 > in2) : in1 ? in2"
       _in="float" _out="float"/>
```

Listing 1.8. The *Max* component of Listing 1.5 replaced with an instantiation of the *binary_ operator* template

While this form of parametric templates is an improvement over the complete absence of a similar feature in XMI or HUTN, parameter passing through string substitution can only serve relatively simple use-cases. To enable more complex reuse scenarios, we have prototyped support for templates where the body is specified using a model scripting and a model-to-text transformation language, instead of XML.

Listing 1.9 shows a more generic implementation of the *binary_ operator* template of Listing 1.6, named *nary_ operator* which can produce components with n input ports (of the same type) and one output port, using the model-to-text EGL [8] language. Line 7 specifies that the body of the template is a model-to-text transformation, while lines 9-14 specify the EGL transformation that will produce the XML of the template instantiation, which will then be parsed by Flexmi.

```
1  <?nsuri comps?>
2  <_>
3    <_template name="nary_operator">
4      <parameter name="n"/>
5      <parameter name="in"/>
6      <parameter name="out"/>
7      <content language="EGL">
8        <![CDATA[
9        <comp>
10         [%for (i in 1.to(n.asInteger())){%]
11         <in name="in[%=i%]" type="${in}"/>
12         [%}%]
13         <out name="result" type="${out}"/>
14       </comp>
15       ]]>
16     </content>
17   </_template>
18 </_>
```

Listing 1.9. The binary-operator template expressed using an embedded model-to-text transformation in EGL

Clearly, there are several things that can go wrong with template definition and instantiation: templates can return elements that are incompatible with their application context, they can contain syntax errors or they can throw exceptions at runtime. At this early stage, the prototype implementation does not attempt to prevent any of this exceptional behaviour; this is left as future work. Flexmi, including the features discussed in this paper, is available as part of the latest interim version of the Epsilon open-source model management platform (`eclipse.org/epsilon`).

4 Conclusions and Future Work

This paper has presented extensions to the language-agnostic Flexmi textual concrete syntax and parser that aim at enhancing modularity and reuse. The *import* and *include* processing instructions are straightforward and stable extensions for splitting models over multiple files. The templating mechanism discussed in Section 3.3 is more novel, interesting and underdeveloped - and hence a clear direction for additional work.

References

1. Object Management Group. Human-Usable Textual Notation Specification, 2004. `http://www.omg.org/spec/HUTN/1.0/`.
2. Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. Constructing Models with the Human-Usable Textual Notation. In Krzysztof Czarnecki, Ileana Ober, Jean-Michel Bruel, Axel Uhl, and Markus Völter, editors, *Model Driven Engineering Languages and Systems*, pages 249–263, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.
3. Object Management Group. XML Metadata Interchange (version 2.5.1), 2015. `http://www.omg.org/spec/XMI/2.5.1/`.
4. Moritz Eysholdt and Heiko Behrens. Xtext: Implement Your Language Faster Than the Quick and Dirty Way. In *Proceedings of the ACM International Conference Companion on Object Oriented Programming Systems Languages and Applications Companion*, OOPSLA '10, pages 307–309, New York, NY, USA, 2010. ACM.
5. Holger Krahn, Bernhard Rumpe, and Steven Völkel. MontiCore: a framework for compositional development of domain specific languages. *International Journal on Software Tools for Technology Transfer*, 12(5):353–372, Sep 2010.
6. L. Bettini. The EMF Parsley DSL for developing EMF applications. In *4th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 301–308, 2016.
7. Dimitrios S. Kolovos, Nicholas Matragkas, and Antonio García-Domínguez. Towards Flexible Parsing of Structured Textual Model Representations. In *Proceedings of the 2nd Workshop on Flexible Model Driven Engineering co-located with ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2016), Saint-Malo, France, October 2, 2016.*, pages 22–31, 2016.
8. Louis M. Rose, Richard F. Paige, Dimitrios S. Kolovos, and Fiona A. C. Polack. The epsilon generation language. In Ina Schieferdecker and Alan Hartman, editors, *Model Driven Architecture – Foundations and Applications*, pages 1–16, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.