White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/208475/

Version: Published Version

Article:
Kolovos, Dimitris and de la Vega, Alfonso (2023) Flexmi: a generic and modular textual syntax for domain-specific modelling. Software and Systems Modeling. pp. 1197-1215. ISSN: 1619-1366

https://doi.org/10.1007/s10270-022-01064-3

Reuse
This article is distributed under the terms of the Creative Commons Attribution (CC BY) licence. This licence allows you to distribute, remix, tweak, and build upon the work, even commercially, as long as you credit the authors for the original work. More information and the full terms of the licence here:
https://creativecommons.org/licenses/

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

UNIVERSITY OF LEEDS
University of Sheffield
UNIVERSITY of York

Software and Systems Modeling (2023) 22:1197–1215
https://doi.org/10.1007/s10270-022-01064-3

REGULAR PAPER

FLEXMI: a generic and modular textual syntax for domain-specific modelling

Dimitris Kolovos[1] · **Alfonso de la Vega**[2]

Received: 24 February 2022 / Revised: 14 October 2022 / Accepted: 31 October 2022 / Published online: 18 November 2022
© The Author(s) 2022

Abstract
Domain-specific languages allow engineers and domain experts to express problems and design solutions using domain-focused vocabularies and abstractions, by means of graphical or textual syntaxes. In the case of textual syntaxes, language engineers can opt for creating a language-specific syntax by defining and maintaining a BNF-style grammar, or use an existing general-purpose reflective syntax such as the XML Metadata Interchange (XMI) or the Human Usable Textual Notation (HUTN), which do not require any development and maintenance effort, but which are more verbose and cannot be customised. We present FLEXMI: a new general-purpose textual syntax for defining models that conform to Eclipse Modelling Framework's Ecore-based metamodels. FLEXMI offers XML and YAML/JSON syntax flavours, it can be fuzzily parsed to reduce verbosity, and it includes a templating system to facilitate encapsulation of reusable composite model element structures, thus enabling more concise model specifications. We have evaluated FLEXMI for verbosity and model loading performance against XMI, HUTN, and a bespoke (i.e. custom) textual syntax for Ecore (Emfatic). Our results indicate that the use of fuzzy parsing and templates allow FLEXMI to achieve a significant reduction in the verbosity of models compared to XMI/HUTN and can become almost as concise as a bespoke textual syntax, with a moderate performance penalty.

Keywords Domain-specific languages · Generic textual syntaxes · Model-driven engineering

1 Introduction

Model-based software development processes commonly involve the development of domain-specific languages (DSLs) that provide domain-focused vocabularies and abstractions. A DSL consists of an abstract syntax describing the language's concepts, features and permitted relationships; one or more concrete (e.g. graphical, textual) syntaxes that allow modellers to construct models conforming to the language; and artefacts specifying the language's semantics (e.g. denotational/operational semantics specifications, translators, interpreters). A DSL is classified as *internal*, when it extends the syntax and semantics of an existing language, or *external*, when its abstract and concrete syntaxes are defined from scratch.

This paper is concerned with text-based editing of models conforming to *external* DSLs, defined using the metamodelling facilities of the widely used and open-source Eclipse Modelling Framework (EMF) [1]. In this technological space, developers of DSLs have two main options for textual model editing: they can either define a dedicated grammar for their textual syntax using tools such as Xtext [2], EMFText [3], or Monticore [4], from which language-specific parsers and editors can be generated; or they can use generic syntaxes such as HUTN [5], EMFJSON [6] or XMI [7].

The main benefits of language-specific textual syntaxes are customisability and conciseness. On the flip side, developing a bespoke textual syntax requires substantial expertise and upfront development effort, and incurs a maintenance cost over time as the abstract syntax of the language and the underlying framework/IDE evolve. General-purpose reflec-

Communicated by Zhenjiang Hu.

The work presented in this paper has been funded through the HICLASS InnovateUK project (Contract No. 113213).

✉ Alfonso de la Vega
alfonso.delavega@unican.es

Dimitris Kolovos
dimitris.kolovos@york.ac.uk

[1] Department of Computer Science, University of York, York, UK

[2] Software Engineering and Real-Time Group, Universidad de Cantabria, Santander, Spain

tive syntaxes on the other hand are more verbose and lack in customisability, but they eliminate the cost and effort of developing, distributing and maintaining language-specific tooling. Intuitively, reflective syntaxes can be more desirable at the early stages of development of a DSL where the abstract syntax is fluid or for smaller-scale applications that do not justify the effort of implementing and maintaining custom tooling, while bespoke syntaxes can be more appropriate at more mature stages of development and in larger endeavours where the return of investment is clearer.

To address some of the weaknesses of existing reflective textual syntaxes, we present FLEXMI: a new XML/YAML-based, general-purpose textual syntax for EMF-based DSLs, which (1) can be fuzzily parsed to reduce verbosity, and (2) includes a templating mechanism for encapsulating reusable composite model element structures to reduce repetition.

To assess FLEXMI's verbosity and repetition reduction capabilities, we replicated 503 Ecore models from a publicly available dataset [8] and we compared the resulting FLEXMI files against equivalent models in XMI, HUTN and in a bespoke textual syntax for Ecore (Emfatic [9]). Our results indicate that FLEXMI is able to reduce model size roughly by half with respect to XMI, with close results to those of Emfatic. These improvements in conciseness come at the cost of lower parsing speed, which depends on the FLEXMI features that are used to specify a model (e.g. reusable templates and the YAML flavour take longer to parse).

Early versions of FLEXMI's fuzzy parsing algorithm and its mechanism for defining reusable templates were introduced in two (workshop) papers, [10,11] respectively. This paper consolidates and updates those works, including the following novel contributions:

- A more detailed commentary on the advantages and disadvantages of bespoke and generic modelling syntaxes in Sect. 2
- A description-by-example of FLEXMI's syntax and features in Sect. 3 that complements the explanation of the parsing algorithm from [10] (adapted in Sect. 3.1)
- A new YAML syntax flavour for FLEXMI models with full feature parity with respect to the original XML flavour (Sect. 3.2)
- Support for dynamic values via executable attributes, and for importing external EOL [12] operations (Sects. 3.5 and 3.8)
- The ability to set attribute values with the contents of external files (Sect. 3.6)
- An improved templating mechanism over the previous work [11] that allows filling elements generated from templates through internal slots (Sect. 3.7)
- A systematic experimental evaluation of the impact of FLEXMI's fuzzy parsing and templating mechanism on verbosity and repetition (Sect. 4)
- A systematic performance evaluation of FLEXMI's XML and YAML-based parsers

The rest of the paper is organised as follows: Section 2 introduces to domain-specific modelling and to bespoke and generic textual syntaxes. Section 3 presents the main features of FLEXMI, and Sect. 4 describes the evaluation we carried out. Lastly, Sect. 5 concludes the article and outlines future work.

2 Background and motivation

2.1 Domain-specific modelling

This paper focuses on external domain-specific languages defined using object-oriented metamodelling techniques [13]. When defining a new modelling language using these techniques, there is a separation between the specification of the intrinsic concepts and relationships that define the language, denoted as the *abstract syntax*; and the graphical or textual constructs that can be used to create models in the language, known as the *concrete syntax*. The abstract syntax is specified with a metamodel, which is generally depicted in the form of a UML-like class diagram containing the main classes of the language, their attributes, and their references. Models that adhere to the structural rules defined in the abstract syntax of a language are said to *conform* to such abstract syntax.

The Eclipse Modelling Framework (EMF) [1] is the de-facto metamodelling standard in the Java ecosystem. EMF's Ecore is an object-oriented meta-metamodelling language that allows defining the abstract syntax of domain-specific languages. As an example of an abstract syntax of a DSL, which is also used to demonstrate FLEXMI throughout the paper, we present a contrived EMF-based project scheduling language (PSL), whose Ecore metamodel is depicted in Fig. 1. In PSL, projects contain tasks, which are allocated to one or more people in a team. Tasks have a start month and a duration (also in months), and people can specify a list of skills and allocate part of their time to a task (*percentage* in class *Effort*).

Creating PSL models would require one or more concrete syntaxes, which allow users to specify models in some sort or graphical or textual manner that is later parsed into an in-memory model conforming to the abstract syntax of Fig. 1. We focus on textual concrete syntaxes in this work, which are introduced next.

2.2 Textual concrete syntaxes

An important component of an external domain-specific language is its concrete syntax, through which users of the

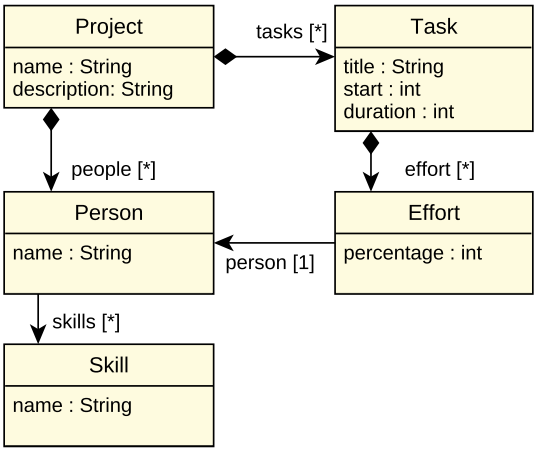

Fig. 1 PSL metamodel in Ecore

```
1  Project {
2      title: "ACME"
3      description: "A bespoke ERP
           System."
4      tasks: Task {
5          title: "Analysis"
6          start: 1
7          duration: 3
8          effort: Effort {
9              person: Person "Alice"
10         }
11     }
12     people: Person "Alice" {
13         name: "Alice"
14     }
15 }
```

Listing 1 Instance of the PSL metamodel in HUTN

language can create models that conform to it. The concrete syntax of a DSL can be graphical, textual, form-based, table-based or hybrid, integrating multiple styles for different parts of the language (e.g. a graphical syntax for the structural parts of the language and an embedded textual micro-syntax for capturing behaviour).

The concrete syntax of a DSL is driven by a range of factors, such as the nature and purpose of the DSL and the skills and preferences of its target audience. In this work, we are concerned with textual syntaxes for external DSLs, with a particular focus on the widely adopted open-source Eclipse modelling ecosystem [1]. Therefore, a discussion on other forms of concrete syntaxes and a comparison of their relative strengths and weaknesses is beyond the scope of this paper.

In the Eclipse modelling ecosystem, there are two main options for textual editing of models that conform to Ecore-based DSLs: defining a bespoke (i.e. custom, specific) textual syntax, or using a generic syntax. These options are discussed in detail as follows:

2.2.1 Bespoke textual syntaxes

Xtext [2,14] is currently the most widely used and actively maintained framework for defining bespoke textual syntaxes for DSLs in the Eclipse modelling ecosystem. From an EBNF-like grammar, the Xtext tooling can produce an ANTLR-based parser for the syntax as well as Eclipse-based and web-based editors that support syntax highlighting, and advanced features such as context-aware code completion, reference navigation and refactoring. Xtext also provides a reusable expression language (Xbase [15]), support for defining custom scoping and name resolution rules, and a workspace indexer. The XSemantics [16] framework extends Xtext with support for defining complex type systems. Frameworks with similar aims and capabilities include EMFText [3], Spoofax [17] and Monticore [4].

The main benefits of developing a bespoke textual syntax using one of the available frameworks discussed above include the ability to fully customise the syntax of the language to meet the needs and accommodate the preferences of its target audience, and the ability to offer high-quality supporting tooling to users with features such as code writing assistance and reference navigation with minimal effort.

On the other hand, a bespoke textual syntax needs to be accompanied by supporting documentation and examples, it needs to co-evolve with the abstract syntax of the language, and mechanisms need to be provided to allow installing and updating its supporting parser and development tools as the abstract syntax, underlying framework (e.g. Xtext) and host IDE (e.g. Eclipse) evolve over time.

2.2.2 Generic textual syntaxes

If developing a bespoke textual syntax is not desirable or is deemed unlikely to provide an acceptable return of investment, another option is to use a generic textual syntax such as the Object Management Group's Human Usable Textual Notation (HUTN) [5], EMFJSON [6] or XMI [7].

An excerpt of a HUTN model that conforms to the PSL metamodel is shown in Listing 1. When the HUTN parser processes this model, it interprets the *Project* token of line 1 as an instance of the *Project* type from the PSL metamodel, it populates the *title* and *description* attributes of the project from the string literals in lines 2 and 3; and it processes the rest of the document in a similar fashion to create, populate and link a task (line 4), an effort (line 8) and a person (line 12) instance.

The XMI and EMFJSON syntaxes are very similar to HUTN, with the main difference being that XMI is XML-based while EMFJSON is JSON-based. All three formats require exact matching between tokens in the textual model and names of classes and features in the metamodel; for example if "description" was changed to "desc" in line 3

Table 1 Pros and Contras of bespoke and generic syntaxes

Syntax type	Pros & Contras
Bespoke	+ As customisable as needed
	+ Can be accommodated to end users
	– Requires some initial cost
	– Requires maintenance
Generic	+ No upfront cost to use them
	– High rigidity due to exact matching
	– Verbose
	– Lacks the ability to reuse elements

of Listing 1, the HUTN parser would fail, as it would not be able to find a feature with that name in the *Project* class. This can make models rather verbose when longer class/feature names are used in the metamodel. Also, none of these syntaxes provides support for encapsulating and reusing (instead of repeating) recurring model element patterns, which are discussed later in Sect. 3.7 of the paper.

Based on the above, bespoke and existing generic syntaxes present some advantages and disadvantages, which we have summarised in Table 1. The following section introduces our flexible and generic syntax that aims to improve the current state of the art.

3 FLEXMI

FLEXMI is a generic textual syntax for EMF-based DSLs which attempts to address the weaknesses listed in Sect. 2.2.2 by providing the following novel features:

- Intelligent and forgiving parsing that does not require exact lexical correspondence with type or feature names present in the metamodel (useful for conciseness)
- A language-agnostic mechanism for defining and instantiating reusable model element templates (useful for conciseness and reuse)

Figure 2 shows an overview of the main components involved in writing and parsing a FLEXMI model. As the following sections explain, FLEXMI offers two syntax flavours: one is XML-based, while the other is based in YAML. Both flavours can be used from the same FLEXMI editor. XML FLEXMI models are translated with a standard parser into an XML DOM, while for YAML a custom parser is used to achieve the same task. Lastly, the FLEXMI parser is able to convert the XML DOM into an in-memory EMF model by applying fuzzy parsing, among other features. Next section describes the FLEXMI parsing process in detail.

```
1  <?nsuri psl?>
2  <proj title="ACME">
3      <desc>A bespoke ERP system</
           desc>
4      <person name="Alice"/>
5      <person name="Bob"/>
6      <task title="Analysis" start="1
           " dur="3">
7          <effort person="Alice"/>
8      </task>
9      <task t="Design" start="4" dur=
           "6">
10         <effort person="Bob"/>
11     </task>
12     <task t="Implementation" start=
           "7" dur="3">
13         <effort person="Bob" perc="
               50"/>
14         <effort person="Alice" perc
               ="50"/>
15     </task>
16 </proj>
```

Listing 2 FLEXMI model conforming to the PSL metamodel of Fig. 1

3.1 Fuzzy parsing of elements

Listing 2 presents an XML document that FLEXMI can parse into a valid instance of the PSL metamodel in Fig. 1. To start using FLEXMI, the only requirement is the existence of an Ecore metamodel. On every FLEXMI model, we must indicate the namespace URI of the metamodel being instantiated: this is achieved by means of a special *nsuri* XML processing instruction. In line 1 of Listing 2, a *nsuri* is used to specify that the model instantiates types from the PSL metamodel. Other XML processing instructions allow importing external FLEXMI models, as well as operations defined in the Epsilon Object Language (EOL) [12] that can be invoked inside the expressions of executable model element attributes (described in Sect. 3.5).

The parsing of a FLEXMI model takes place by transforming XML elements into elements of an in-memory EMF model in a depth-first fashion. An overview of how the XML syntax is used to represent EMF models in FLEXMI is given in Fig. 3. In the following, we discuss how the FLEXMI parser interprets the document of Listing 2. The parsing process is depicted in Fig. 4, and the complete algorithm that is applied over each XML element can be found in Listing 3.

The parsing of XML elements takes place with the help of a stack. When FLEXMI encounters the *proj* element in line 2 of the model, the parser stack is empty, which means that the name of this element is matched against the names of all classes in the PSL metamodel. This case is covered by lines 2–6 of the algorithm in Listing 3. By contrast to parsers of existing reflective textual formats, FLEXMI's fuzzy parser does not require exact lexical matching between the

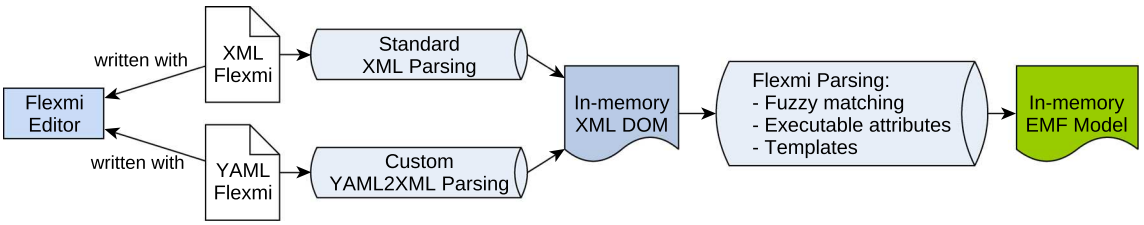

Fig. 2 Overview of the main FLEXMI components and parsing process

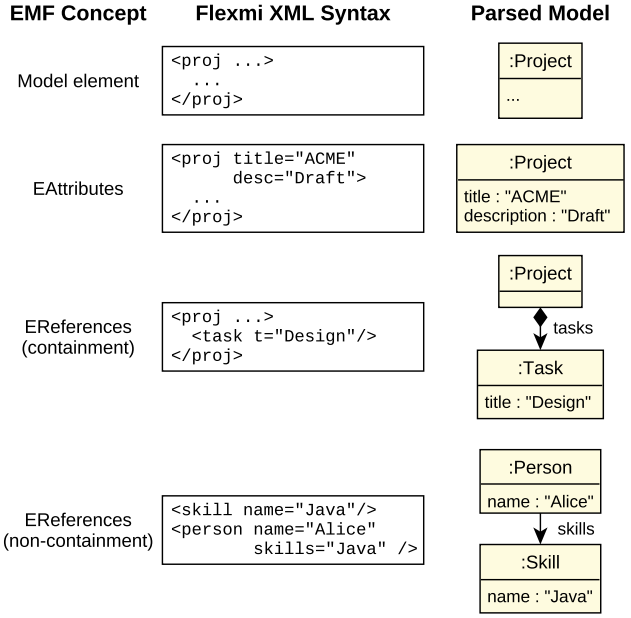

Fig. 3 XML syntax to EMF model in FLEXMI

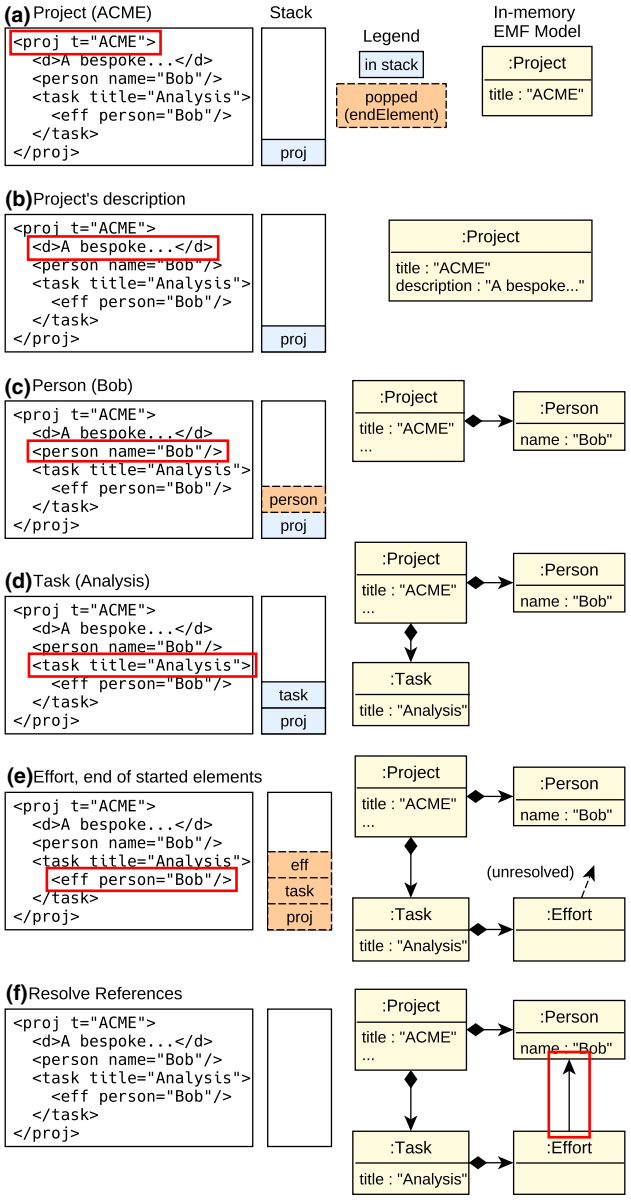

Fig. 4 Parsing process over a simplified Listing 2

names of types/features in the metamodel and the XML tags it encounters in a model—the closest match by name similarity is selected instead. For the *proj* element, this match is the *Project* class, and therefore, FLEXMI creates an instance of *Project* in the in-memory representation of the model. In addition, the project instance is pushed into the stack, to keep track of the context of the children of the *proj* element (Fig. 4a). A model element is maintained in the stack until all descendants of its associated XML element have been processed. For instance, element *desc* in line 3 is only matched against the possible features (i.e. attributes and references) of the *Project* class, and in this case, it is paired with the *description* attribute (Fig. 4b). The structure of this element is treated in a special way by the FLEXMI algorithm (covered in lines 9–12 of Listing 3), because it has no XML attributes, and it only contains text (i.e. no extra children). When that is the case, the (trimmed) text of the XML element is set as the value of the selected feature, which in this example is the project's description. This use of the text of an XML element is useful for multi-line string attribute values.

Line 4 presents the general case of finding an XML element that will eventually be parsed as a model element, while there are model elements in the stack (corresponds with lines 16–30 in Listing 3 and Fig. 4c). Again in the context of a

```
1  procedure start_element(xml_element, stack)
2      if the stack is empty then /* xml_element is interpreted as a model element */
3          let new_model_element_type = the EClass whose name has the highest similarity to
               the tag of the xml_element, chosen from the EClasses in the available
               metamodel(s)
4          let new_model_element = new instance of new_model_element_type
5          call set_attribute_values (new_model_element, xml_element)
6          push new_model_element to the stack
7      else if stack.peek() is a model element then
8          let parent_model_element = stack.peek()
9          if xml_element has no attributes and only text then /* xml_element -> feature */
10             let feat = the feature of the parent_model_element's class with the highest
                   string similarity to the name of the xml_element
11             let parent_model_element.feat = textual content of the xml_element, casting
                   appropriately
12             push null to the stack
13         else if xml_element has no attributes then /* xml_element -> containment slot */
14             let reference = the containment reference of the parent_model_element's class
                    with the highest string similarity to the tag of the xml_element
15             push a containment slot encapsulating parent_model_element and the reference
                   to the stack
16         else /* the element has attributes, so xml_element -> model element */
17             let candidates = all parent_model_element's containment references, and the
                   class types and subtypes of such references
18             let best_candidate = the class or reference with the highest similarity with
                   the name of xml_element
19             if best_candidate is a class
20                 let new_model_element_type = best_candidate
21                 let reference = the one with the highest string similarity to xml_element
                        containing the new_model_element_type among its accepted types
22             else /* best_candidate is a reference */
23                 let new_model_element_type = the class type with the highest string
                       similarity with xml_element among the possible subtypes of
                       best_candidate
24                 let reference = best_candidate
25             end
26             let new_model_element = new instance of new_model_element_type
27             call set_attribute_values (new_model_element, xml_element)
28             add new_model_element to the reference // attending reference multiplicity
29             push new_model_element to the stack
30         end
31     else if stack.peek() is a containment slot then /* xml_element -> model element */
32         let containment_slot = stack.peek()
33         let new_model_element_type = the class with the highest similarity to the tag of
               the xml_element among the subtypes of the reference of the containment_slot
34         let new_model_element = new instance of new_model_element_type
35         call set_attribute_values (new_model_element, xml_element)
36         add new_model_element to the containment_slot's reference // attending
               multiplicity
37         push new_model_element to the stack
38     end
39 end procedure
```

Listing 3 Procedure that is applied to parse each new XML element

project, the parser has two options for interpreting the *person* element: it can be either a *Task* belonging to the *tasks* containment (analogous to UML's composition) reference of the project, or a *Person* belonging to its *people* reference. Based on string similarity, it opts for the latter. Things change in line 6, where the *task* element is a better match for a *Task* under the *tasks* reference. The rest of the XML elements are processed into model elements in the same fashion.

Fuzzy matching is also used to map XML attribute names to attributes and non-containment references of model elements. The algorithm that is used to set element feature values from XML attributes is defined in Listing 4. This algorithm is called in lines 5, 27 and 35 of Listing 3. For example, in line 8 of Listing 2, after a *Task* element is created, the FLEXMI parser uses the Hungarian algorithm [18] to decide the optimal mapping between the *title*, *start* and *dur* XML attribute names and the possible features of elements of type

```
1  procedure set_attribute_values(
       xml_element, model_element)
2    let attrs = the attributes of
       xml_element
3    store variable attributes for later
       resolution and remove them from
       attrs
4    let assignment = the map from attrs to
       features of the model_element type
       with the maximum total similarity
5    foreach xml_attribute in attrs do
6      let feature = the corresponding
         feature of xml_attribute in
         assignment
7      if xml_attribute starts with ":" then
8        /* executable attribute: evaluated
             later */
9        store feature computation of
           xml_attribute, feature and
           model_element
10     else
11       let value = that of xml_attribute
12       if feature is an EAttribute then
13         if feature is single-valued then
14           set feature to value //
               typecast
15         else
16           let values = split value by
               comma
17           set feature to values //
               typecast
18         end
19       else // the feature is an
           EReference
20         if feature is single-valued then
21           create an unresolved reference
22         else
23           let values = split value by
               comma
24           foreach value in values do
25             create an unresolved
                 reference
26           end
27         end
28       end
29     end
30   end
31 end procedure
```

Listing 4 Algorithm that maps XML element attribute values to values of EAttributes and non-containment EReferences

Task (line 4 of Listing 4 and Fig. 4d). In this case, the XML attributes are allocated and hence used to populate the *title*, *start* and *duration* Task attributes, respectively (lines 12–18 of Listing 4).

3.2 YAML/JSON syntax flavours

Apart from XML, FLEXMI also offers a YAML [19] flavour to specify models. The rationale behind this inclusion was providing an even more concise and human-readable syntax on top of what is already achieved by the fuzzy parsing of XML-based FLEXMI models.

```
1  ?nsuri: psl
2  project:
3    - name: ACME
4    - desc: A bespoke ERP system
5    - person: {name: Alice}
6    - person: {name: Bob}
7    - task:
8      - title: Analysis
9      - start: 1
10     - dur: 3
11     - effort:
12       - person: Alice
13   - task: {
14     title: Design,
15     start: 4,
16     dur: 6,
17     effort: {person: Bob}
18   }
19   - task: {
20     title: Implementation,
21     start: 7,
22     dur: 3,
23     effort: {person: Bob, perc: 50},
24     effort: {person: Alice, perc:
         50}
25   }
```

Listing 5 FLEXMI model of Listing 2 using the YAML flavour

Listing 5 shows the FLEXMI model of Listing 2 represented in YAML. In this format, content is organised as entries, composed of a key and a value separated by a colon (:). There are two main constructs to specify the nesting structure of these entries: block sequences and flow mappings. In YAML, block styles use space-based indentation to denote structure, while flow styles include explicit indicators to organise entries. Both types of styles can be mixed to define the elements of a model.

Block sequences are formed by placing a succession of entries at the same level of indentation and by having each entry key prefixed by a dash (-) and a space. For instance, lines (7–12) of Listing 5 define a block sequence that represents the attributes and references of the *Analysis* task, and line 12 (which is indented an extra level) represents a single-entry sequence containing a *person* reference that belongs to the *effort* element of line 11. Alternatively, flow mappings are specified by surrounding with braces a set of comma-separated entries. In the example, lines 13–18 show a specification of the *Design* task that uses a flow mapping to define the task features. This flow style basically allows defining models using the JSON [20] format, and this is possible due to YAML being a superset of JSON.

Implementation-wise, and to minimise duplication, YAML-based FLEXMI models are parsed into an XML Document Object Model (DOM), and then, they are processed by the algorithms shown in Listings 3 and 4 as it happens with XML-based models. Figure 5 depicts model examples of how the

main YAML constructs are translated into XML, and how the final XML is then parsed to obtain an EMF model. Examples *a* and *b* show two YAML specifications of the same PSL model, composed of a root element (a *project*), an attribute (the project's *name*), and a nested element (a *task*). While example *a* uses block sequences for structuring elements, example *b* applies flow mappings. Nevertheless, the internal XML DOM generated during the parsing of both examples is the same, which is shown as example *c*. These examples also show an *nsuri* processing instruction being detected and translated: any key of a YAML entry starting with a question mark (?) symbol is translated into a processing instruction in the internal XML DOM. Lastly, examples *d* and *e* show a model fragment where an *Alice* person is defined having two skills: Java and XML. These examples present the last YAML constructs required when specifying FLEXMI models: block sequences of scalars and flow sequences. Both constructs are used to specify lists of scalars (i.e. primitive values), which in EMF are needed to specify non-containment references (e.g. the skills of the person in the example) and multi-valued attributes. While block sequences imply placing each scalar into its own line (example *d*), flow sequences are delimited by brackets and separate the scalars by commas, like JSON arrays (example *e*). Example *f* shows the resulting translation to XML of *d* and *e*, where the scalar values are stored as the text of XML elements. These text elements are processed by FLEXMI as described in Sect. 3.1 and in lines 0–12 of Listing 3.

The FLEXMI parser detects whether a file is XML- or YAML-based by checking the first non-whitespace character of the file contents: when it is a less-than symbol (<), FLEXMI selects XML as the model flavour; it selects YAML otherwise. This detection also makes it possible for the Eclipse-based FLEXMI editor to detect the flavour used and adapt its syntax highlighting capabilities accordingly.

Both the XML and YAML syntax flavours have full feature parity, so they can specify the same FLEXMI models. Apart from personal preference, using a more human-readable and writeable syntax is the main reason for choosing the YAML flavour over the XML one. On the flip side, XML might be a better candidate when models have multi-line text attributes, as these can be specified as the text content between the tags of an XML element, such as the *description* of a PSL *Project* (line 3 of Listing 2). Other reasons for choosing XML over YAML are avoiding mandatory style requirements like the space indentation required in Fig. 5a, d, or the minor parsing performance penalty of having to first parse the YAML models into XML DOMs.

For simplicity, the examples found in the remainder of this article are presented using the XML flavour. The interested reader can find alternative model specifications in YAML as part of the external repository accompanying this work.[1]

```
1  <?nsuri psl?>
2  <proj title="ACME">
3   <regular-tasks>
4       <task t="Analysis" start="1"
            dur="3"/>
5     <task t="Design" start="4" dur="
          6"/>
6     </regular-tasks>
7     <dissemination-tasks>
8       <task t="Seminar" start="6"
            dur="2"/>
9   </dissemination-tasks>
10 </proj>
```

Listing 6 Example of containment reference slots to disambiguate in which reference to store tasks

3.3 Containment slots

Depending on the modelling language, a model element could fit into more than one containment reference of its parent. For instance, let's suppose that a PSL project has two containment references of type *Task*, namely *regularTasks* and *disseminationTasks*, instead of the *tasks* reference depicted in Fig. 1. Naming XML elements that represent tasks as *task* (e.g. as in Listing 2) would not explicitly indicate in which of the two references the element should be placed. The FLEXMI parser would choose a reference based on name similarity,[2] which in this case would mean that *tasks* elements would always be placed into *regularTasks*. There are two ways to actually specify which containment reference to populate. First, instead of using *task* to name the task elements, we could use a name closer to the containment reference we wish to select, e.g. *regtask* or *dissemtask*, or the full reference name if preferred. Second, FLEXMI offers an optional construct to include model elements (e.g. tasks) under the compatible containment reference of our choice: containment slots.

Listing 6 shows an example with two containment slots: *regular-tasks* and *dissemination-tasks*. A containment slot is detected by FLEXMI because of having a name with the highest string similarity with a containment reference and having no XML attributes (lines 13–15 of Listing 3). When such a containment slot is detected, it is pushed to the parser's stack, so any children of the slot are directly added to the associated containment reference of the parent model element (lines 31–37 of Listing 3). Containment slots might be preferred

[1] Click to visit an external repository containing YAML and XML representations of the paper examples.

[2] FLEXMI currently uses Levenshtein's distance [21] to calculate string similarity, but other methods could also be applied.

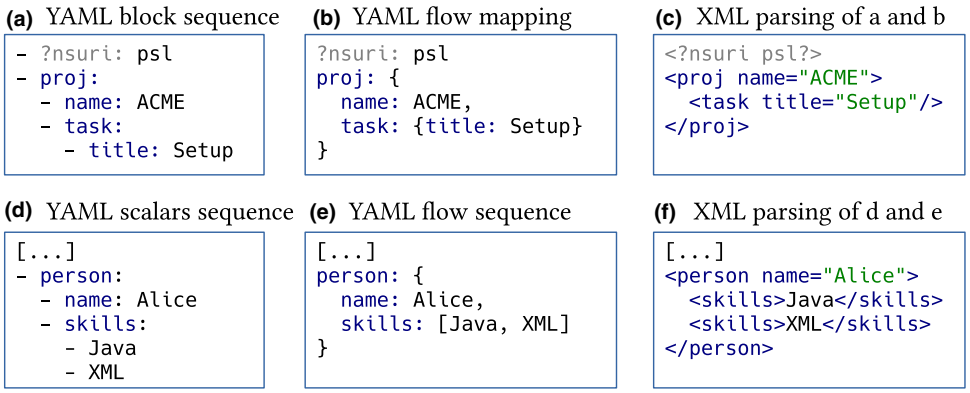

Fig. 5 Parsing of YAML representations using block and flow-based styles to XML DOMs

over the alternative of using the containment reference name when users want to be consistent with the naming of model elements, e.g., using *task* to denote all Task elements of the model.

3.4 Non-containment references

So far we have discussed how the FLEXMI parser interprets XML element names and attributes to create model elements and populate their containment references and attributes. To support non-containment references such as the *person* reference of the PSL *Effort* type, target elements need to have a unique identifier. If a class has an attribute marked as *identifier* in the Ecore metamodel [1], FLEXMI will use that to identify its instances. Otherwise, it will use the value of the *name* attribute, if present. Using this convention, the XML element of Listing 2, line 7, which is interpreted as an instance of the *Effort* type, refers to the *Alice* person defined in line 4 of the document via her name. Fully qualified ID paths separated by dots (.) are also supported. The path of an element is formed by combining the IDs of all its containers until the root of the model is reached (i.e. *ACME.Alice* to refer to the person in line 4). These paths can be useful to resolve ambiguities caused when two elements have the same local identifier. Independently of the ID that is finally used, non-containment references are collected during the parsing process (Listing 4, lines 19–27 and Fig. 4e), and they are resolved at the end of the document, so that all elements that can be referenced are already present in the in-memory model (Fig. 4f).

It could happen that a model element has neither an attribute defined as identifier in its associated class, nor a name attribute that could be used instead. In this case, it is still possible to reference this element through the use of FLEXMI's variables and executable attributes, which are introduced next.

```
1  <?nsuri psl?>
2  <project title="ACME">
3  <person name="Alice" :global="alice
       "/>
4  <person name="Bob"/>
5
6  <task title="Analysis" start="1"
       dur="3">
7  <effort person="Alice"/>
8  </task>
9  <task t="Design" start="4" dur="6"
       :var="design">
10 <effort person="Bob"/>
11 </task>
12 <task title="Implementation" :start
       ="design.start + design.duration
       " dur="3">
13 <effort person="Bob" perc="50"/>
14 <effort :person="alice" perc="50"/>
15 </task>
16 </project>
```

Listing 7 Using variables and executable attributes

3.5 Executable attributes and variables

Prepending a colon (:) to the name of an attribute instructs the FLEXMI parser to interpret its value as an executable EOL [12] expression instead of a literal value. Also, FLEXMI supports attaching a *:var* or a *:global* attribute to XML elements, to declare local/global variables that can be used in EOL expressions. The scope of local (*:var*) variables includes siblings of the element, and their descendants, while global variables can be accessed from anywhere in the model.

For example, in line 9 of the FLEXMI model in Listing 7, the *Design* task is assigned to a local variable named *design*, which is then used in line 12 to compute the value of the start month of the *implementation* task.

Continuing with the ways of targeting elements from the previous section, line 3 of Listing 7 shows how the person instance can be made available through a *:global* attribute denoted *alice* (notice the lowercase name). That attribute creates a global variable, which is later used in line 14 to refer to

```
1  <?nsuri psl?>
2  <proj title="ACME"
3       desc_="acmeDescription.txt">
4    ...
5  </proj>
```

Listing 8 Loading the description of a project from the *acmeDescription.txt* file

the person element contained in such variable through an executable *:person* attribute. As in the case for non-containment references, variables and executable attributes are collected and resolved once the end of the document has been reached (lines 3 and 9 of Listing 4).

3.6 Setting attribute values from file contents

Apart from using executable attributes as defined above, it is also possible to set model element features with the contents of external files. To do this, the name of the XML attribute must have an underscore (_) suffix, and the value of the attribute should point to the file from which to load the contents. The loading of the contents of the referenced file as the value of the XML attribute takes place before calling to the *start_element* algorithm of Listing 3, which allows using fuzzy matching for the attribute name as if it were a regular one.

Listing 8 shows how the description of a project is loaded from an external file *acmeDescription.txt*, by using the *desc_* attribute name and the relative path of the file, which indicates that the text file must be located in the same folder as the FLEXMI model. In the different places where FLEXMI can reference external resources, both absolute/relative filesystem paths and Eclipse-based platform URIs are supported.

3.7 Templates

FLEXMI supports defining reusable templates through the reserved `<:template>` XML tag. For example, when designing one-person projects where all tasks take place in sequence, we can omit all the repetitive `<effort>` elements that refer to the same person, and we can automate the calculation of the start date of each task by using a template, as shown in Listing 9.

All FLEXMI templates have two properties: the *name* that must be used as tag name to instantiate the template; and a *content* children, which provides the XML elements that will be processed by the FLEXMI parser. In Listing 9, a template named `simpletask` is defined in lines 10–21. This template is used three times in lines 5–7. Based on its content (lines 11–20), each usage of this template generates a *task* element, which also contains an *effort* element. The start of the task and the person allocated to the effort are defined

```
1  <?nsuri psl?>
2  <_>
3      <project title="ACME">
4          <person name="Alice"/>
5          <simpletask title="Analysis
               " dur="3"/>
6          <simpletask title="Design"
               dur="3"/>
7          <simpletask title="Impl."
               dur="6"/>
8      </project>
9
10     <:template name="simpletask">
11         <content>
12         <task :start="Task.all
13       .indexOf(self)
14       .asVar('index') == 0 ? 1 :
15           Task.all.get(index-1)
16               .asVar('previous').
                     start
17               + previous.duration">
18             <effort :person="Person
                   .all.first"/>
19         </task>
20         </content>
21     </:template>
22 </_>
```

Listing 9 FLEXMI model with a `simpletask` template

through executable attributes. The *:start* expression calculates the start time of a task by obtaining the task index in the list of all existing tasks, and then by accumulating the duration of all preceding tasks. The *:person* expression simply assigns the effort to the first Person element of the model (i.e. the only one available, *Alice*). As a side note, this is the first appearance of the `<_></_>` special XML element, which is used to support FLEXMI models that have more than one top-level elements (similarly to the `<xmi:xmi>` tag in XMI).

3.7.1 Parametric templates

FLEXMI templates also support string parameters, which can be used to customise the generated content when instantiated. Listing 10 includes a template (`solo`) that can be used to define tasks carried out by a single person (lines 8–16). This template accepts a *name* and a *person* parameter (lines 9 and 10), which are used to name the task and to assign all the effort to the provided person, respectively. Parameters are provided as regular XML attributes when instantiating the template, as it happens in line 5 where *Design* and *Alice* are passed as *name* and *person*.

```
1  <?nsuri psl?>
2  <_>
3      <project title="ACME">
4          <person name="Alice"/>
5          <solo name="Design" dur="3"
               person="Alice"/>
6      </project>
7
8      <:template name="solo">
9          <parameter name="name"/>
10         <parameter name="person"/>
11         <content>
12             <task name="${name}">
13                 <effort person="${
                       person}"/>
14             </task>
15         </content>
16     </:template>
17 </_>
```

Listing 10 A template (`solo`) with *name* and *person* parameters

```
1  <?nsuri psl?>
2  <_>
3      <project title="ACME">
4          <person name="Alice"/>
5          <longtask title="
               Implementation" years="2
               ">
6            <effort person="Alice"/>
7          </longtask>
8      </project>
9
10     <:template name="longtask">
11         <parameter name="years"/>
12         <content language="EGL">
13         <![CDATA[
14         <task dur="[%
15        <:slot/>
16         </task>
17         ]]>
18         </content>
19     </:template>
20 </_>
```

Listing 11 A dynamic template (`longtask`) with a `<:slot>`

3.7.2 Dynamic templates and slots

It is possible to use embedded model-to-text transformations to further customise the content that templates produce. In Listing 11, an EGL [22] transformation is defined inside the `<content>` element of the `longtask` template (lines 10–19). This template can be used to create tasks using years as duration unit, by providing a *years* value through a parameter (lines 5 and 11), which is translated to months in the EGL code (line 14). Also, FLEXMI supports including a `<:slot>` element in the content of templates, which specifies where any nested elements of the caller (e.g. the effort element of line 6) should be placed in the produced XML.

```
1  <?nsuri psl?>
2  <_>
3      <person name="Alice"/>
4      <person name="Bob"/>
5      <person name="Charlie"/>
6  </_>
```

Listing 12 The `people.flexmi` model

```
1  <?nsuri psl?>
2  <?import people.flexmi?>
3  <proj title="ACME">
4      <task title="Analysis" start="1
           " dur="3">
5          <effort person="Alice"/>
6      </task>
7  ...
```

Listing 13 Importing `people.flexmi` in a model

3.8 Importing other FLEXMI Models and External Operations

Other FLEXMI models can be imported through the use of the `<?import other.flexmi?>` and `<?include other.flexmi?>` processing instructions. The use of *import* creates a new EMF Resource for the `other.flexmi` file, which is useful for referencing elements and for having the same FLEXMI model imported by several models. On the other hand, *include* parses the contents of `other.flexmi` as if they were embedded in the FLEXMI model that contains the *include* processing instruction, just as the *input* command works for embedding LaTeX documents. This inclusion happens at the position where the processing instruction is placed, this means, we could include the contents of an external FLEXMI file as children of a concrete XML element in the source file.

In the context of PSL, all *Person* elements working in different projects could be centralised in a model denoted `people.flexmi`, which is depicted in Listing 12. This model can be imported from FLEXMI models containing project details to reference the people that will carry out that project tasks, as shown in Listing 13. While we could also use the *include* instruction here, using *import* is preferred because if any project model also references another project, then the `people.flexmi` will only be loaded once.

It is also possible to use processing instructions to import operations contained in an external EOL file, with the aim of using them as part of executable attributes (see Sect. 3.5). For instance, the expression of the *:start* attribute in the `simpletask` template of Listing 9 (lines 12–17) is a one-liner that could be made more readable if divided into several lines. Listing 14 shows a `pslOperations.eol` file doing just that in the *getStartTime()* operation, which is defined in

```
operation Task getStartTime() {
    var taskIndex = Task.all.
        indexOf(self);
    if (taskIndex == 0) {
        return 1;
    } else {
        var previous = Task.all.get
            (taskIndex-1);
        return previous.start +
            previous.duration;
    }
}
```

Listing 14 The pslOperations.eol file

```
<?nsuri psl?>
<?eol pslOperations.eol?>
<_>
    ...
    <:template name="simpletask">
    <content>
    <task :start="self.getStartTime
        ()">
        <effort :person="Person.all
            .first()"/>
    </task>
    </content>
    </:template>
</_>
```

Listing 15 Updates Listing 9 to use the external pslOperations.eol file

a block of instructions and with the help of syntax highlighting. Then, Listing 15 shows how to use this EOL file from a FLEXMI model: the *eol* processing instruction must be used (line 2), which allows changing the *:start* attribute to a simpler call to *getStartTime()* (line 7).

3.9 Tool support

FLEXMI is developed as part of the Eclipse Epsilon project,[3] and it is supported by a dedicated Eclipse editor that can be seen on Fig. 6. As mentioned in Sect. 3.2, this editor can detect whether the chosen flavour for the opened FLEXMI model is XML or YAML, and then select the appropriate syntax highlighting. In addition, the editor offers comprehensive error reporting capabilities, including malformed XML/YAML mistakes, exceptions in executable attributes, unresolved references, missing imported/included files, or other errors detected by standard EMF validation (e.g. violation of minimum and maximum metamodel cardinalities, omission of mandatory features). In Fig. 6, the editor (top right) shows the FLEXMI model depicted in Listing 2. This editor is integrated with other views of the Eclipse IDE. For instance, the top left of the figure shows the Outline view,

[3] https://www.eclipse.org/epsilon/.

```
ResourceSet resourceSet =
    new ResourceSetImpl();
resourceSet.
    getResourceFactoryRegistry()
    .getExtensionToFactoryMap()
    .put("flexmi", new
        FlexmiResourceFactory());
Resource resource =
    resourceSet.createResource(
        URI.createFileURI("acme.
            flexmi"));
resource.load(null);
```

Listing 16 Java snippet that loads a FLEXMI model as an EMF Resource

which depicts the tree structure of the in-memory EMF model parsed from the FLEXMI model. The elements of this EMF model can be inspected in the Properties view (bottom left), which in the figure shows the three attributes of the *Analysis* task selected in line 6 of the editor. Lastly, any warnings or errors in the model would be listed in the Problems view (bottom right), as well as marked in the editor. In the example, we have introduced a small mistake by assigning the string *fifty* to the *percentage* numerical attribute of the *effort* element in line 13. The Problems view shows this mistake as a warning, including its location in the `acme.flexmi` file.

FLEXMI also provides an implementation of EMF's *Resource* interface, which allows FLEXMI models to be consumed by any EMF-compatible application or model management language (e.g. ATL, Acceleo). Listing 16 shows a Java snippet that loads a FLEXMI model as a standard EMF Resource. Additionally, FLEXMI offers a facility for transforming FLEXMI models to XMI. Finally, apart from being installable as an Eclipse bundle, FLEXMI is also available as a standalone Java library on Maven Central[4] and is used as the modelling format of choice in Epsilon's web-based Playground.[5]

3.10 Limitations

Although FLEXMI models can be seamlessly loaded and used by any EMF-compatible application, changes made to their in-memory representations cannot be serialised back to XML/YAML. While the shortened terms of the fuzzy matching could be stored and recovered for serialisation, the results of executable attributes and the application of templates (see Sects. 3.5 to 3.7) cannot be unrolled in the general case.

In Sect. 3.1, we introduced the *nsuri* processing instruction that allows specifying the namespace URIs of the metamodels that the FLEXMI model is instantiating. Multiple *nsuri*

[4] https://mvnrepository.com/artifact/org.eclipse.epsilon/org.eclipse.epsilon.flexmi.

[5] https://eclipse.org/epsilon/live.

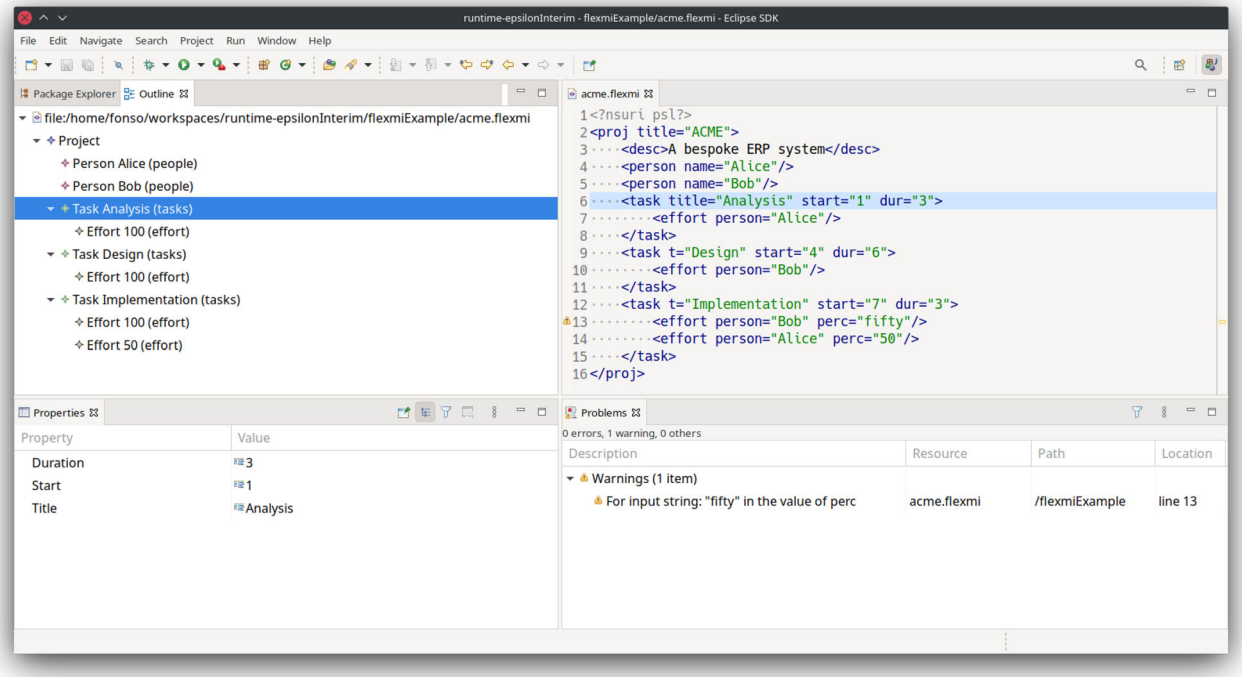

Fig. 6 The FLEXMI editor and its integration with the Outline, Properties and Problems view from the Eclipse IDE

instructions can appear in a FLEXMI model if it contains instances of types from multiple metamodels. However, a caveat of FLEXMI's fuzzy matching mechanism is that it is not able to differentiate between two classes with the same name coming from different metamodels. While this issue could be solved by adopting namespace prefixes, this option was discarded in favour of simplicity and conciseness, as supporting such prefixes would bring back part of the undesirable complexity and verbosity of XMI.

Metamodel evolution could cause unexpected mappings of XML elements or attributes to model element types or features, respectively. For instance, if an Effort's *percentage* attribute is renamed to *cost* in the PSL metamodel of Fig. 1, the FLEXMI parser will automatically map the *perc* XML attributes in lines 13–14 of Listing 2 to the new *cost* attribute of an Effort, which might or might not be the desired behaviour. While setting a minimum similarity threshold below which no matching is accepted could help, this is an approach that needs to be further studied, as establishing a threshold that is too restrictive could also prevent valid matches from being accepted.

Lastly, as part of the tooling, FLEXMI's Eclipse-based editor does not currently offer syntax highlighting for inlined code from Epsilon languages, such as EOL expressions (e.g. see Sect. 3.5) or EGL-based dynamic templates (described in Sect. 3.7.2). We will aim to support these languages as future work.

4 Evaluation

To evaluate FLEXMI, we measured the impact of fuzzy parsing and templates in terms of conciseness and performance. All code and artefacts involved in this evaluation can be found in an external repository.[6] In terms of testing, the FLEXMI parser is backed by 57 automated unit and integration tests that assert that it behaves as expected against 76 test models, and protect future development from regressions.

4.1 Evaluation method

We compared FLEXMI against three existing textual syntaxes for Ecore models: XMI and HUTN (generic syntaxes), and Emfatic (bespoke syntax). For the purposes of our evaluation, we treat Ecore as an EMF-based object-oriented DSL (i.e. a mini UML), overlooking its role as the metamodelling language of EMF. The reasons behind opting for Ecore are (1) the availability of many existing Ecore models in the public domain (to avoid bias), and (2) the availability of existing bespoke textual syntaxes for it such as Emfatic, Xcore [23] and OCLInEcore [24]. The selection of Emfatic over Xcore and OCLInEcore was a free choice given that all three syntaxes are very similar in terms of conciseness.

[6] https://github.com/alfonsodelavega/flexmi-evaluation.

4.1.1 Ecore model dataset

We reused the dataset presented in [8], which consists of 2,420 XMI-based Ecore models mined from different open source software repositories. However, most of these Ecore models contain issues, such as syntactical errors or unresolvable proxies (i.e. references to external models). We limited the evaluation to self-contained models (i.e. no proxies), which did not suffer from errors during the evaluation procedure, and with at least 5KiB in size (to filter out toy examples), ending up with 503 models, whose XMI byte size ranges from 5KiB to 3.4 MiB.

4.1.2 Textual syntaxes generation

The Ecore models in the dataset were stored in XMI, from which we automatically generated replica models in the other syntaxes. For instance, for Emfatic and HUTN, we used the built-in transformations provided by their implementations to obtain, for each Ecore XMI model, an Emfatic and HUTN-based model, respectively.

We used a model-to-text transformation to generate FLEXMI Ecore models out of the XMI ones. In particular, we generated two FLEXMI model versions for each Ecore model: one that makes use of templates (see Sect. 3.7) and one that does not (denoted as *plain* in the following). The rationale behind this decision was so that we could independently measure the conciseness benefits and performance overhead of the templating mechanism on top of a plain version of FLEXMI, where only fuzzy parsing was applied (described in Sect. 3.1). Additionally, we generated two FLEXMI models for each variant: one using the XML flavour, and another using YAML (introduced in Sect. 3.2), with the objective to measure the overhead of the extra YAML-to-XML transformation that takes place when using the YAML flavour. Therefore, a total of four FLEXMI models was generated for each Ecore XMI model.

The plain FLEXMI versions only benefit from fuzzy matching mechanisms to allow a concise wording of Ecore terms for the model tags and attributes. Table 2 shows how these terms were shortened when generating FLEXMI models. The shortened terms were not chosen focusing exclusively on reducing length, as this might make the model more difficult to understand (i.e. using *u* instead of *upperBound* for ETypedElements), but also in maintaining readability (so, *upper* was finally used in this case).

The templated versions are plain FLEXMI models that also use a set of templates where possible. An excerpt of the templates file for XML FLEXMI models can be seen in Listing 17. These templates are mostly parametric ones (see Sect. 3.7.1), and allow representing Ecore *EAttributes* (lines 3–9), *EReferences* (lines 10–24), and a special type of *EAnnotations* (GenModel documentation, in lines 25–32) more concisely.

Table 2 Fuzzy terms shortened in FLEXMI Ecore models

XMI term	Shortened as
EPackage	package
EClass	class
eStructuralFeatures (for EAttributes)	attr
eStructuralFeatures (for EReferences)	ref
eType (ETypedElement)	type
upperBound (ETypedElement)	upper
lowerBound (ETypedElement)	lower
EAnnotation	annotation
EOperation	op
EDataType	type
EEnum	enum
EEnumLiteral	lit

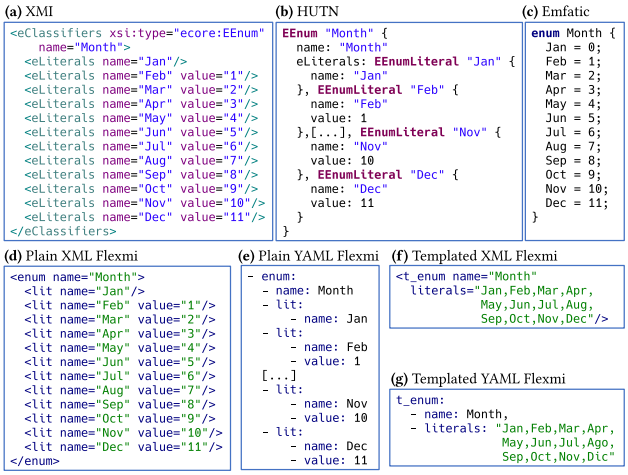

Fig. 7 An Ecore EEnum representing the months of the year in all the compared notations

Applying templates allowed us to mimic the bespoke Emfatic syntax in some cases. For instance, there are two templates that use the *val* term to represent containment references, just as Emfatic does. The *val* term is used for single-valued containment references (lines 10–14), while *vals* is used for multi-valued ones (lines 15–19). There is also a dynamic template *t_enum* (see Sect. 3.7.2) that allows representing an Ecore's *EEnum* in a single XML element that enumerates the list of literal names, instead of having to define a tag for the enum and then an additional nested element for each literal (*EEnumLiteral*). This is achieved by means of a model-to-text transformation (lines 36–45), which iterates over the list of literals to generate the *EEnumLiteral* elements in the background, assigning an incremental literal value to each one of them. Figure 7 contains an example of a *Month* EEnum expressed in all the compared syntaxes, which were extracted from one of the metamodels of the dataset .

```
<?nsuri psl?>
<_>
<!-- There are extra templates that match
     attributes with other Ecore EDataTypes -->
<:template name="string">
    <content>
        <attr type="//EString"><:slot /></attr>
    </content>
</:template>
<:template name="val">
    <content>
        <ref containment="true"><:slot /></ref>
    </content>
</:template>
<:template name="vals">
    <content>
        <ref containment="true" upper="-1"><:slot/></ref>
    </content>
</:template>
<:template name="refs">
    <content>
        <ref upper="-1"><:slot/></ref>
    </content>
</:template>
<:template name="genmodel">
    <parameter name="doc"/>
    <content>
        <annotation source="http://www.eclipse.org/emf/2002/GenModel">
            <details key="documentation" value="${doc}"/>
        </annotation>
    </content>
</:template>
<:template name="t_enum">
    <parameter name="name"/>
    <parameter name="literals"/>
    <content language="EGL">
   <![CDATA[
   <eenum name="[%
     <:slot/>
     [%
     <lit name="[%
     [%
   </eenum>
   ]]>
  </content>
</:template>
</_>
```

Listing 17 Some of the FLEXMI XML templates for Ecore

4.1.3 Measuring method

For the conciseness, we measured the character counts of the models corresponding to each syntax, omitting whitespace. This size measure was preferred over a more conventional lines of code (LOC) one because of the different styles of the compared syntaxes, e.g. the tag-based format of the XMI and FLEXMI XML models differs from the Java-like syntax of Emfatic. Nonetheless, given the popularity of the LOC measure we also included it in our analysis.

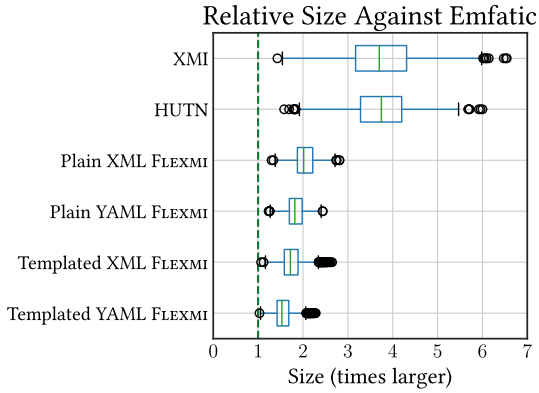

Fig. 8 Character counts relative to Emfatic (dashed line)

Related to performance, we compared model loading times of each textual syntax. Due to the small size of some models, individual model loading measurements turned impractical, as some load operations took less than one millisecond. Therefore, we instead measured the accumulated time it took each syntax to load the whole dataset of models, which was in the order of seconds.

Performance measurements were carried out in a MacBook Pro with a quad-core i5 CPU, 32 GiB of LPDDR4X RAM, and an NVME SSD. To increase reliability, measurements were taken 20 times, and unmeasured warm-up repetitions were included to mitigate any perturbation due to idle states of the operating system.

4.2 Results

4.2.1 Conciseness comparison

To compare measurements of heterogeneous models, character counts were normalised with respect to the ones of Emfatic. As this notation offers a tailored syntax for Ecore, it consistently achieved the lowest character counts across all measured models. Therefore, the closer the results of other syntaxes get to the ones of Emfatic, the better they score in terms of conciseness. Figure 8 shows character count box plots of XMI, HUTN, and the four FLEXMI variants relative to the Emfatic results, which are represented by a green dashed line at the "1" value.

At a first glance, there is a higher dispersion in the XMI and HUTN results with respect of those of the FLEXMI versions, whose boxes and distances between the exterior whiskers are smaller. This higher dispersion can be explained by these syntaxes having a much greater verbosity for certain syntax constructs with respect to Emfatic, and by the different proportion of these constructs in the models of the dataset.

(a) XMI

```
<eStructuralFeatures
  xsi:type="ecore:EAttribute"
  name="path"
  eType="ecore:EDataType
    http://www.eclipse.org/emf/
    2002/Ecore#//EString"/>
<eStructuralFeatures
  xsi:type="ecore:EReference"
  name="contents"
  upperBound="-1"
  eType="#//Content"
  containment="true"/>
```

(b) HUTN

```
eStructuralFeatures:
  EAttribute "path" {
    name: "path"
    eType: "ecore:EDataType",
      "http://www.eclipse.org/
      emf/2002/Ecore#//EString"
  }, EReference "contents" {
    name: "contents"
    upperBound: -1
    eType: "#//Content"
    containment: true
  }
}
```

(c) Plain XML Flexmi

```
<attr name="path"
      type="//EString"/>
<ref name="contents"
     upper="-1"
     containment="true"
     type="Content"/>
```

(d) Templated XML Flexmi

```
<string name="path"/>
<vals name="contents"
      type="Content"/>
```

(e) Emfatic

```
attr String path;
val Content[*] contents;
```

Fig. 9 Examples of how EAttributes and EReferences are represented in the compared notations

(a) XMI

```
<eAnnotations source="http://www.eclipse.org/emf/2002/GenModel">
    <details key="documentation" value="Added in 4.0"/>
</eAnnotations>
```

(b) Plain XML Flexmi

```
<annotation source="http://www.eclipse.org/emf/2002/GenModel">
    <details key="documentation" value="Added in 4.0"/>
</annotation>
```

(c) Templated XML Flexmi

```
<genmodel doc="Added in 4.0"/>
```

(d) Emfatic

```
@GenModel(documentation="Added in 4.0")
```

Fig. 10 GenModel annotation representation

For instance, Ecore *EAttributes* and *EReferences* take way more characters to be expressed in XMI and HUTN than in Emfatic. An example of this can be seen in Fig. 9. We can see that the Emfatic syntax (e) is free of all the verbosity that is required for the XMI and HUTN serialisation (a and b, respectively). Therefore, models having a greater number of *EReferences* and *EAttributes* would have a greater relative size in XMI/HUTN with respect to Emfatic than those containing a lower proportion of these syntax constructs. Similar comparisons can be made with other Ecore syntax constructs, such as *EAnnotations* or *EEnums* (see Fig. 7).

While the same analogy can be made for the FLEXMI versions, these are much more concise than XMI, making their results less sensitive to the input model contents and thus more consistent. Coming back to Fig. 9 example, in plain FLEXMI (c) an *attr* or *ref* tag name is enough to start the element, and similar fuzzy names are used for tag attributes (see Table 2 for other fuzzy matching examples). When defining attribute types, FLEXMI allows omitting the Ecore namespace URI, so just the type identifier needs to be specified, which in the case of Ecore is done through a name-based URI (e.g. *//EString*). Lastly, as depicted in Fig. 9d, the templated FLEXMI version includes templates for data types and for representing containment references of different multiplicities. In the example, an *EAttribute* of type *EString* is represented simply with a *string* tag, and a multi-valued containment reference is defined by using the *vals* tag (these FLEXMI templates are defined in Listing 17, among others). Although Fig. 9 only contains the FLEXMI examples for the XML flavour due to space restrictions, the same rationale applies to the YAML one too, which in addition is a bit more concise because of the added verbosity of XML tags.

When considering numerical results, the XMI and HUTN median values sit at 3.70 and 3.74, respectively, which are considerably higher than the 2.01 and 1.81 results of the XML and YAML plain FLEXMI versions, and than the 1.71 and 1.53 results of the templated versions, respectively. Focusing on the 75^{th} percentile (i.e. the rightest line of the box of each boxplot), XMI and HUTN sizes are 4.31 and 4.20 times those of Emfatic, while the FLEXMI versions are only 2.21, 1.98, 1.88 and 1.68 times as big.

Lastly, related to LOC measurements, the effect of certain Ecore syntax constructs is even more pronounced than in the character counts, specially in the case of EAnnotations. We show an example of this in Fig. 10 depicting a GenModel annotation, which is used to include documentation in model elements. Such an annotation always has, at least, a nested *doc* detail element. As can be seen, this annotation takes a minimum of 3 LOC in XMI. In plain XML FLEXMI, it takes the same number of lines, while there is a GenModel template that can be seen in lines 25–32 of Listing 17, so the templated version of FLEXMI can make use of it to take only one LOC. Emfatic has a specific syntax construct for annotations that allows representing them in a single line as well. Moreover, this syntax construct can be also used with custom annotations, these are, those that were defined specifically for a concrete Ecore metamodel, in which case there is no generic FLEXMI defined in our evaluation setup. Therefore, Emfatic can represent any EAnnotation in the metamodel as a one-liner.

In summary, Emfatic has a great advantage in the LOC measurements, and a model-by-model relative comparison as done in the character count analysis was not as fair in this case. Therefore, we performed a dataset-level LOC count, answering the following question: *how many lines does it take each syntax to specify all Ecore models in the dataset?* The results can be found in Table 3.

Starting with HUTN and the FLEXMI YAML versions, these syntaxes require more LOC because of their greater usage of new lines when specifying model elements. When comparing the two, HUTN requires a greater number of lines, due to the extra curly brace that is required to close each element (e.g. see lines 10–11 or 14–15 of Listing 1). With respect to the XML-based syntaxes, we can see how $\sim 10\%$ of

Table 3 Number of LOC required to represent the whole dataset of Ecore models

Syntax	#LOC
HUTN	789,430
Plain Yaml FLEXMI	692,546
Templated YAML FLEXMI	536,044
XMI	351,027
Plain XML FLEXMI	314,053
Templated XML FLEXMI	250,156
Emfatic	167,521

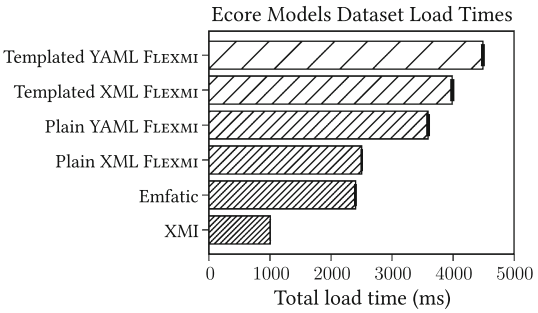

Fig. 11 Accumulated time taken to load the whole dataset

lines are skipped in plain XML FLEXMI with respect to XMI, which goes up to ∼ 28% when FLEXMI templates are also applied. Lastly, and mainly due to the special EAnnotation syntax described above, Emfatic is able to use ∼ 52% less lines than XMI.

4.2.2 Performance comparison

Figure 11 shows the accumulated time it took the Emfatic, XMI and FLEXMI parsers to load the whole models dataset, including 95% confidence intervals. We opted to leave HUTN out of these results, as the performance of its implementation [5] is not on par with the others (the full dataset load took more than 8 minutes, far from the seconds it takes the other approaches). We can see that the results for Emfatic and plain XML FLEXMI are very close, at 2.39 and 2.49 times the results of XMI (i.e. the faster notation), respectively. As for the templated XML FLEXMI version, its result is 3.96 times the one of XMI, and 1.5 times that of Emfatic. This is due to the overhead of processing of parametric and dynamic templates (see Sects. 3.7.1 and 3.7.2, respectively). The FLEXMI YAML flavours are the slower ones, taking 3.57 times that of XMI for the plain version, and 4.46 times for the templated version. When comparing the flavours, the plain YAML version takes 30% more time than the XML one, while the templated YAML version takes 12% more time than the XML counterpart. This increase was expected, as the YAML models are first transformed to an XML DOM before being processed by the FLEXMI parser.

Table 4 Some statistics of the Ecore models in the dataset

Measurement	#Classes	#Attributes	#References
First Quartile	11	11	11
Median	20	19	19
Third Quartile	39	35	43
Max	945	7933	881

4.3 Threats to validity

We comment here on any detected threat that might be influencing the outcome of our experiments.

With the aim of avoiding bias in the results for the specific case of Ecore, XMI, HUTN and Emfatic, we opted for a third-party dataset with a reasonable distribution in terms of the size and key characteristics of the models it contains, such as the number of classes, attributes and references in each, summarised in Table 4.

It could also be argued that using just an example language (Ecore) may not be enough to generalise the results. While this might be true, we prioritised quality over quantity of examples, this is, we avoided the creation of synthetic examples to prevent the potential inclusion of biases due to the performed experiments. In fact, the inherent simplicity of the Ecore language caused that only a handful of templates were worth considering, so some FLEXMI features might prove even more useful when applied to models from other domains, which we will explore in the future.

The better results in conciseness provided by FLEXMI might come at a cost of readability of the models. We tried to mitigate this risk by selecting terms that offered a good balance between conciseness and understandability (see Table 2 to check how terms were shortened). When in doubt, the complete, original terms were used. We will perform real readability and learnability experiments with end users as part of our future work.

For the comparison, the models of the two FLEXMI versions and of the Emfatic notation were automatically generated. So, it could be argued that the conciseness results are tied to how good the generator that created the models is. After a manual inspection of the automatically generated models, apart from some whitespace (which is ignored in the character counts of the comparison), we did not detect any extra-verbose syntax construct that might be improved if manually defined. We consider that model contents would be very close to those of hand-crafted models if the transformation was performed manually instead (a much more error-prone task though), so we believe that the use of

```
<or>
    <and>
        <var id="a"/>
        <var id="b"/>
    </and>
  <var id="c"/>
</or>
```

Listing 18 Boolean expression captured in FLEXMI

transformations to obtain the compared models of certain notations is not affecting the validity of the results.

The incurred performance overhead ranged from 2.5 to 4.5 slower parse times with respect to XMI, being in some configurations nearly as fast to parse as the bespoke Emfatic syntax. On the other hand, the performance of a bespoke parser can depend on a number of factors such as the sophistication of the underlying parser generator and the complexity of the BNF grammar, and therefore, the performance results against Emfatic cannot be safely generalised.

4.4 Discussion

While FLEXMI is certain to be more concise than XMI and HUTN, and has been shown to be nearly as concise as Emfatic, bespoke textual syntaxes can be substantially more concise than FLEXMI in some scenarios. For example, in the context of a DSL that allows defining arbitrarily complex Boolean expressions, a bespoke syntax could provide a very concise encoding such as *(a and b) or c*, while it would require a much more verbose encoding in FLEXMI as shown in Listing 18.

In general, factors that can affect the compactness of a custom textual syntax include (1) the ability to reuse established concise notations that the target audience are already familiar with (e.g. single-character mathematical symbols with already well-understood semantics instead of longer keywords) and (2) the training effort one is prepared to invest since a more concise syntax might require more training for users to understand and remember.

Beyond conciseness, when deciding whether FLEXMI or a custom syntax is more appropriate for the task at hand, the following concerns should be taken into account:

- Stability of the metamodel: in early iteration cycles FLEXMI can be preferable to a custom syntax as it can eliminate the need to co-evolve a grammar as the metamodel evolves;
- Expected return of investment: a custom textual syntax can provide usability benefits but also involves developing, maintaining, testing and distributing dedicated software (e.g. a grammar, scoping rules). While for large-scale projects (e.g. long-lived, many developers) the usability benefits of dedicated tooling can justify this additional effort, for smaller-scale projects a generic textual syntax like FLEXMI can be preferable.

5 Conclusions and future work

We have presented FLEXMI, a generic textual syntax for EMF-based DSLs that provides greater conciseness, flexibility and customisability than existing generic syntaxes, while avoiding the upfront cost of developing a bespoke textual syntax. FLEXMI achieves this by applying fuzzy parsing, by allowing the definition and instantiation of reusable templates, and by supporting dynamic functionality through executable expressions and embedded model-to-text transformations. It also provides two feature-equivalent syntax flavours to choose from, based in XML and YAML, respectively.

Our planned future work includes improving the FLEXMI editor to better support syntax highlighting of inlined EOL/EGL languages, auto-completion and preview/navigation of references. We also wish to evaluate the performance implications of making the parsing algorithm smarter by e.g. also considering value types during attribute allocation, or by including look-ahead mechanisms that check deeper levels of the parsed DOM to decide e.g. whether an XML element represents a model element or a containment reference slot. Usability and learnability experiments involving end users trying out FLEXMI are also part of our future goals.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

References

1. Steinberg, D., Budinsky, F., Paternostro, M., Merks, E.: EMF: Eclipse Modeling Framework, 2nd edn. Addison-Wesley Professional (2009)
2. Eysholdt, M., Behrens, H.: 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, pp. 307–309 (2010)

3. Heidenreich, F., Johannes, J., Karol, S., Seifert, M., Wende, C.: Model-based language engineering with emftext. In: International Summer School on Generative and Transformational Techniques in Software Engineering, pp. 322–345. Springer (2011)
4. Hölldobler, K., Rumpe, B.: MontiCore 5 Language Workbench Edition 2017. Aachener Informatik-Berichte, Software Engineering, Band 32. Shaker Verlag (2017). http://www.se-rwth.de/phdtheses/MontiCore-5-Language-Workbench-Edition-2017.pdf
5. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.: Constructing models with the human-usable textual notation. In: Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28–October 3, 2008. Proceedings, Lecture Notes in Computer Science, vol. 5301, pp. 249–263. Springer (2008). https://doi.org/10.1007/978-3-540-87875-9_18
6. Hillairet, G.: EMFJSON—EMF Binding for JSON. https://github.com/emfjson/emfjson-jackson
7. Group, O.M.: XML metadata interchange (XMI) specification. https://www.omg.org/spec/XMI/About-XMI/
8. Barriga, A., Di Ruscio, D., Iovino, L., Nguyen, P.T., Pierantonio, A.: An extensible tool-chain for analyzing datasets of metamodels. In: Proceedings of the 23rd ACM/IEEE International Conference on Model Driven Engineering Languages and Systems: Companion Proceedings, pp. 1–8. ACM, Virtual Event Canada (2020). https://doi.org/10.1145/3417990.3419626
9. Foundation, E.: Emfatic. https://www.eclipse.org/emfatic/
10. Kolovos, D.S., Matragkas, N., García-Domínguez, A.: 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 & Systems (MoDELS 2016), Saint-Malo, France, October 2, 2016, CEUR Workshop Proceedings, vol. 1694, pp. 22–31. CEUR-WS.org (2016). http://ceur-ws.org/Vol-1694/FlexMDE2016_paper_3.pdf
11. Kolovos, D.S., Paige, R.F.: Towards a modular and flexible human-usable textual syntax for EMF models. In: R. Hebig, T. Berger (eds.) Proceedings of MODELS 2018 Workshops: ModComp, MRT, OCL, FlexMDE, EXE, COMMitMDE, MDETools, GEMOC, MORSE, MDE4IoT, MDEbug, MoDeVVa, ME, MULTI, HuFaMo, AMMoRe, PAINS co-located with ACM/IEEE 21st International Conference on Model Driven Engineering Languages and Systems (MODELS 2018), Copenhagen, Denmark, October, 14, 2018, CEUR Workshop Proceedings, vol. 2245, pp. 223–232. CEUR-WS.org (2018). http://ceur-ws.org/Vol-2245/flexmde_paper_3.pdf
12. Kolovos, D.S., Paige, R.F., Polack, F.: The epsilon object language (EOL). In: A. Rensink, J. Warmer (eds.) Model Driven Architecture—Foundations and Applications, 2nd European Conference, ECMDA-FA 2006, Bilbao, Spain, July 10–13, 2006, Proceedings, Lecture Notes in Computer Science, vol. 4066, pp. 128–142. Springer (2006). https://doi.org/10.1007/11787044_11
13. Kleppe, A.: Software Language Engineering: Creating Domain-Specific Languages Using Metamodels. Addison-Wesley Professional (2008)
14. Bettini, L.: Implementing Domain-Specific Languages with Xtext and Xtend. Packt Publishing Ltd. (2016)
15. Efftinge, S., Eysholdt, M., Köhnlein, J., Zarnekow, S., von Massow, R., Hasselbring, W., Hanus, M.: Xbase: implementing domain-specific languages for Java. ACM SIGPLAN Not. **48**(3), 112–121 (2012)
16. Bettini, L.: Implementing type systems for the ide with xsemantics. J. Log. Algebr. Methods Progr. **85**(5, Part 1), 655–680 (2016). https://doi.org/10.1016/j.jlamp.2015.11.005. (**Special Issue on Automated Verification of Programs and Web Systems**)
17. Wachsmuth, G.H., Konat, G.D., Visser, E.: Language design with the spoofax language workbench. IEEE Softw. **31**(5), 35–43 (2014)
18. Kuhn, H.W.: The Hungarian method for the assignment problem. Nav. Res. Logist. Q. **2**(1–2), 83–97 (1955). https://doi.org/10.1002/nav.3800020109
19. YAML: YAML Ain't Markup Language. https://yaml.org/
20. JSON: JavaScript Object Notation. https://www.json.org/
21. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Sov. Phys. Dokl. **10**, 707–710 (1966)
22. Rose, L.M., Paige, R.F., Kolovos, D.S., Polack, F.: The epsilon generation language. In: I. Schieferdecker, A. Hartman (eds.) Model Driven Architecture—Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9–13, 2008. Proceedings, Lecture Notes in Computer Science, vol. 5095, pp. 1–16. Springer (2008). https://doi.org/10.1007/978-3-540-69100-6_1
23. Foundation, E.: Xcore. https://wiki.eclipse.org/Xcore
24. Foundation, E.: OCLInEcore. https://wiki.eclipse.org/OCL/OCLinEcore

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Dimitris Kolovos is a Professor of Software Engineering in the Department of Computer Science at the University of York, where he researches and teaches automated and model-driven software engineering. He is also an Eclipse Foundation committer, leading the development of the open-source Epsilon model-driven software engineering platform, and an editor of the Software and Systems Modelling journal. He has co-authored more than 150 peer-reviewed papers, and his research has been supported by the European Commission, UK's Engineering and Physical Sciences Research Council (EPSRC), Innovate UK and by companies such as Rolls-Royce and IBM.

Alfonso de la Vega is an Assistant Professor at the University of Cantabria. Previously, he was a Research Associate working at the University of York. He collaborates as an Eclipse Foundation Committer for the Epsilon project. His more recent research focuses on novel model visualisation and comparison approaches. He has also worked in how to apply modelling and domain-specific languages to reduce the complexity of carrying out data engineering and data mining tasks.