JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Co-evolution of Metamodel and Generators: Higher-order Templating to the Rescue

Tiziano Lombardi, Vittorio Cortellessa, and Alfonso Pierantonio
Università degli Studi dell'Aquila, Italy

ABSTRACT In Model-Driven Engineering, metamodels are the cornerstone entities underpinning modeling environments. Given one or more metamodels, a diversity of artefacts, including models, transformations, and code generators, are formally coupled with them. Like any other software, metamodels are prone to evolutionary pressure due to shifting business requirements. However, metamodel modifications might come at the price of jeopardizing the related artefacts that, in turn, must be adapted in order to remain valid. While a comprehensive corpus of research has shown that the co-evolution of metamodels and models can be effectively addressed with semi-automated techniques, the co-evolution of transformations and code generators still requires significant manual intervention. This paper proposes a novel technique to make template-based code generators resilient, to some extent, against metamodel evolution. A new template-based language, called hotello, is proposed for the specification of *meta-templates*, i.e., higher-order model-to-text transformations. The approach has been implemented, and a demonstration of its capabilities on a case study in the IoT domain is discussed.

KEYWORDS Coupled evolution, Change resilience, Code generators, Templating, Higher-order transformation language.

1. Introduction

In Model-Driven Engineering (Schmidt 2006) (MDE), metamodels are at the core of any modeling environment. Such environments are typically complex, tailored for specific application domains, and made of numerous metamodel-dependant artefacts, whether editors, transformations, analyzers, or code generators. Defining a metamodel is a complex and iterative process that aims to analyze, engineer, and formalize an application domain by involving different stakeholders. Evolution is an inevitable aspect that affects the whole life-cycle of software systems (Lehman & Belady 1985), and metamodels are not exempted from this general law. On the contrary, they are steadily subject to evolutionary pressure due to abrupt requirements, novel insights emerging from the domain, or simply bug-fixing (Di Ruscio et al. 2012). However, modifying a metamodel comes at the price of jeopardizing the corresponding modeling ecosystem because its components might result no longer valid after the changes; therefore, they must be consistently adapted to restore their validity: this is typically referred to as the co-evolution problem (Van Der Straeten et al. 2008).

Over the last years, many approaches for the co-evolution of metamodels and models emerged, e.g., see (Hebig et al. 2016) for a survey. They typically differ in the way they identify metamodel changes and how instances are migrated. Besides the more general problem of (meta)model matching and comparison (Kolovos et al. 2009), programmatic and generative approaches are adopted for performing model migration (Di Rocco et al. 2012). Such techniques have been applied, with different degrees of success, also to the migration of other kinds of artefacts, including model transformations (Wimmer et al. 2010; García et al. 2012; Kusel et al. 2015; Rutle et al. 2020), GMF-based editors (Di Ruscio et al. 2010), and code generators (Di Ruscio et al. 2013; Di Rocco et al. 2014). Unfortunately, spelling out the requirements for a proper migration is not an easy task, and translating captured requirements into correct operational procedures can be even harder regardless of the adopted technique. In particular:

– generative techniques present the advantage of performing

JOT reference format:
Tiziano Lombardi, Vittorio Cortellessa, and Alfonso Pierantonio.
Co-evolution of Metamodel and Generators: Higher-order Templating to the Rescue. Journal of Object Technology. Vol. 20, No. 3, 2021.
http://dx.doi.org/10.5381/jot.2021.20.3.a7

the migration directly from a model-based representation of the metamodel changes as differences; however, differencing algorithms are not always accurate and consistency restoration is based on heuristics encoded in higher-order transformations that make them hardly customizable and manageable;
– programmatic techniques can be more accurate and manageable; however, their scalability is limited due to the difficulty of writing and maintaining refactoring programs that must consistently migrate the considered artefacts or even co-evolve them together with the metamodels.

Arguably, such difficulties limit the ability to keep a modeling environment aligned to shifting (business) demands leading to a potential lock-in in the defined abstractions and generators (Visser et al. 2007).

This paper presents a novel approach that introduces a notion of *resilience* in the co-evolution of metamodels and generators, i.e., model-to-text transformations. Generators are usually defined by means of template-based languages, such as Acceleo[1] or EGL[2]. Given a metamodel, templates are defined to generate textual artefacts (regardless of the target notation) whose content is extracted by querying the conforming models. In order to make, to a certain extent, templates persistent to metamodel changes, this paper proposes hotello, a higher-order template-based language (and the corresponding implementation) for the specification of *meta-templates*. Analogously to a higher-order model transformation, a meta-template is used for generating a class of templates that are, in turn, applied for code generation. The approach is based on the observation that the same syntactic structures in the target notation may originate from ontologically similar information. Thus, as long as metamodel changes are not disruptive and refer to the same kind of information, meta-templates can still operate and produce the expected output despite the different syntax. The required *syntactical tolerance* is provided by a simple annotation mechanism that can categorize the information described in the metamodel for later use in the meta-template. The effectiveness of the approach is demonstrated by applying the proposed techniques in the domain of Internet-of-Things.

Outline. The paper is organized as follows. The next section illustrates a motivating scenario that highlights the need for more *resilient* to change artefacts instruments in code or text generation. Section 3 presents the hotello language and the related approach. In Sect. 4, a case study related to the Internet-of-Things is illustrated, showing how meta-templating provides a robust solution for generating code in the context of evolving metamodels. Related works are discussed and compared in Sect. 5. Finally, conclusions are drawn in Sect. 6 alongside a brief outline of future work.

2. Motivational Scenario

This section illustrates an elementary scenario in which we show how simple modifications to an Entity-Relationship metamodel affect an associated template written in EGL for the generation of SQL queries. Let $\texttt{ER}_1$ be the metamodel in Fig. 1, where a `Diagram` consists of an arbitrary number of `Entities` inheriting the `name` attribute from the `Named` metaclass. Each `Entity` has typed `Attribute`(s) with one of them denoted as (primary) `Key`. An instance of the metamodel is given in Fig. 2, which specifies a `PhoneBook` with the contact details of an `employee`. The EGL

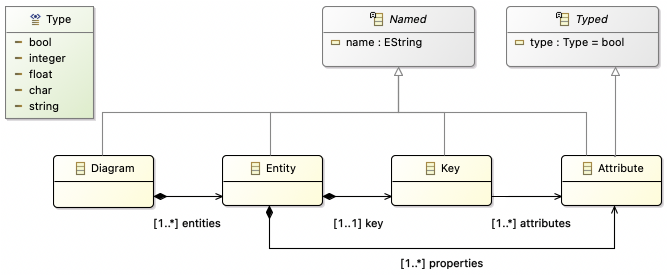

Figure 1 $\texttt{ER}_1$, an Entity-Relationship metamodel

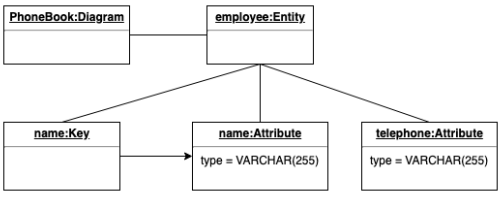

Figure 2 The `PhoneBook` model conforming to $\texttt{ER}_1$

template in Listing 1 defined upon $\texttt{ER}_1$, starting from a conforming model, can generate the corresponding queries for creating the database tables, as follows: for each entity, a `CREATE` query is generated (line 4-5) with a column for each associated property (line 8-9) and a primary key (line 13). Once applied the to the model in Fig. 2, it generates the query in Listing 2 for creating the corresponding table.

It is worth noting that the template contains several references to syntactical elements in $\texttt{ER}_1$. Therefore, if changes involve such elements, then the corresponding template expressions might become invalid and break the consistency with $\texttt{ER}_1$. Let us consider the metamodel $\texttt{ER}_2$ in Fig. 3, as obtained by modifying $\texttt{ER}_1$, where additions are highlighted in green, whereas modifications in red (Ohst et al. 2003). In particular:

– a metaclass called `Relation` is added with `properties` and `entities` as structural features, where the latter one refers to the two entities forming the relation;

```
1  [%import "utils.eol";%]
2
3  [* MS SQL Entity Create Table *]
4  [%for (t in source!Entity.all) {%]
5  CREATE TABLE [%=t.name%] (
6
7      [* Columns *]
8      [%for (c in t.properties) {%]
9      [%=c.name%] [%=c.type.literal%],
10     [%}%]
11
12     [* Primary Key *]
13     PRIMARY KEY ([%=t.key.attributes.namesToList()%])
14 );
15 [%}%]
```

Listing 1 An $\texttt{ER}_1$ template for generating `CREATE` queries

[1] https://www.eclipse.org/acceleo/
[2] https://www.eclipse.org/epsilon/doc/egl/

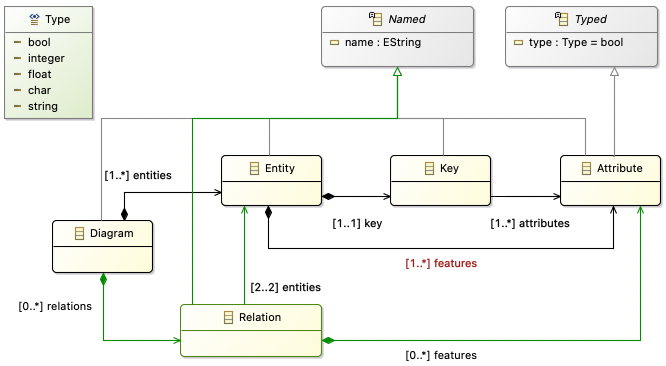

Figure 3 ER_2, a refactoring of ER_1

```
CREATE TABLE employee (
    name VARCHAR(255),
    telephone VARCHAR(255),

    PRIMARY KEY (name)
);
```

Listing 2 The query generated from the `PhoneBook` model

- a `relations` structural feature is added, which collects all the relations in the `Diagram`;
- the `properties` old structural feature is renamed as `features`.

Because of the performed changes, the template illustrated above is not valid anymore[3]. Therefore, the template must be consistently adapted to *a)* deal with the renaming and *b)* consider the newly added modeling elements, e.g., the Relation metaclass, in the query generation. A possible template adaptation is the following[4]:

```
[%import "utils.eol";%]

[* MS SQL Entity Create Table *]
[%for (t in source!Entity.all) {%]
CREATE TABLE [%=t.name%] (

    [* Columns *]
    [%for (c in t.features) {%]
    [%=c.name%] [%=c.type.literal%],
    [%}%]

    [* Primary Key *]
    PRIMARY KEY ([%=t.key.attributes.namesToList()%])
);
[%}%]

[* MS SQL Relation Create Table *]
[%for (t in source!Relation.all) {%]
CREATE TABLE [%=t.name%] (

    [* Extra Columns *]
    [%for (c in t.features) {%]
    [%=c.name%] [%=c.type.literal%],
    [%}%]

    [* Foreign Keys *]
    [%for (k in t.key) { %] [%var c = k.attributes.first();%]
  [%=c.name%] [%=c.type.literal%],
  [%}%]

    [%for (k in t.key) {%]
    FOREIGN KEY ( [%=k.name%] )
          REFERENCES [%=k.eContainer().name%] ( [%=k.name%] ),
    [%}%]
);
[%}%]
```

Listing 3 An ER_2 template for generating `CREATE` queries

Like the previous one, the new template generates a `CREATE` query for each `Entity` instance occurring in a model. A `CREATE` query is generated for each `Relation` as well. Let us consider the model conforming to ER_2 presented in Fig. 4. It contains a version of the `PhoneBook` in which an `employee` can be associated with her `unit` by means of the `belongs` relation. At this point, the application of the new template on the extended `PhoneBook` given place to the following queries

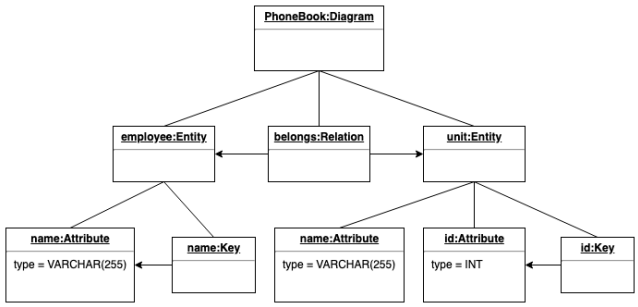

Figure 4 An extended `PhoneBook` model conforming to ER_2

```
CREATE TABLE employee (
    name VARCHAR(255),

    PRIMARY KEY (name)
);
CREATE TABLE Unit (
    id INT,
    name VARCHAR(255),

    PRIMARY KEY (id)
);

CREATE TABLE belongs (
  name VARCHAR(255),
  id INT,

    FOREIGN KEY ( name )
          REFERENCES employee ( name ),
    FOREIGN KEY ( id )
          REFERENCES unit ( id ),
);
```

Listing 4 The queries generated from the extended `PhoneBook` model

Despite the simplicity of the changes operated in the scenario discussed in this section, they had severe repercussions on the considered tooling chain. The query-generating template was urged to be consistently co-evolved in order to keep operational the modeling infrastructure and to avoid a metamodel lock-in. Unfortunately, the maintenance of different artefacts and tools consistent with the metamodel is challenging, and even a simple renaming or extension can force the modeler to opt for leaving the metamodel unchanged.

In the sequel of the paper, we will present an approach to model-to-text transformations that, instead of adapting the related templates, makes them resilient to a certain extent to changes.

3. A higher-order template-based approach

Based on the previous section's considerations, we present a higher-order transformation language, called hotello. The language can be used for the specification of meta-templates, i.e., templates that can be applied to a class of *similar* metamodels to produce traditional instance-level templates for the generation of textual artefacts. For the sake of clarity, we call the former and the latter ones meta-templates and instance-templates, respectively.

[3] For the sake of accuracy, it should be noted that the renaming breaks the conformance of the `PhoneBook` model as well.

[4] Generally, artefacts can be adapted to a metamodel evolution in multiple ways.

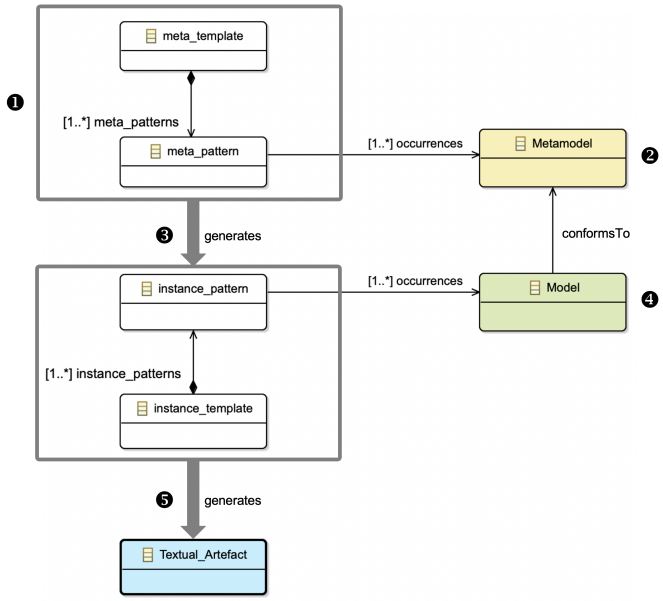

Figure 5 The hotello application architecture

Let us consider the hotello application architecture given in Fig. 5, the meta-template (see ❶) is a textual specification containing a number of so-called *meta-patterns* (besides other syntactical fragments that will compose the target of textual artefacts). A meta-template exemplar is illustrated in Listing 5.

```
1  /* MetaModel-Level used libraries */
2  IMPORT "resources/mylib";
3
4  /* Model-Level used libraries */
5  import "resources/utility";
6
7  /*
8   * This is the rule to create SQL Tables
9   */
10 FORALL (mc: EClass.all TAGGED table) {
11   /* For each matching Metaclass loops through instances */
12   foreach(instance in [mc.name].all) {
13     CREATE TABLE [[instance.name]] #{
14
15       // Table columns
16       FORALL (r: mc.getEReferences() TAGGED column) {
17         /* For each matching reference loops through values */
18         foreach(object in instance.[r.name].properties) {
19           [[object.type]] [[object.name]],
20         }
21       }
22
23       // Table Primary keys
24       FORALL (r: mc.getEReferences() TAGGED primary) {
25         /* For each matching reference loops through values */
26         foreach(object in instance.[r.name].key.attributes.namesToList()) {
27         PRIMARY KEY ( [[object.name]] ),
28         }
29       }
30
31       // Table Foreign keys
32       FORALL (r: mc.getEReferences() TAGGED foreign) {
33         /* For each matching reference loops through values */
34         foreach(object in instance.[r.name].key) {
35         FOREIGN KEY ( [[object.name]] )
36           REFERENCES [[object.parent.name]] ( [[object.name]] ),
37         }
38       }
39     #}
40   }
41 }
```

Listing 5 Meta-template definition for SQL table creation

It refers to the Entity-Relationship domain already presented in Sect. 2 for generating SQL queries. The meta-template contains four meta-patterns (denoted by uppercase and pink colored text) universally quantifying over metaclasses (line 10) and references (lines 16, 24, and 32), respectively. Moreover, only modeling elements with specific annotations (denoted by the `TAGGED` clause) are selected. Now, if we consider the $\texttt{ER}_1$ metamodel on the left-hand side of Fig. 6 in place of the metamodel occurring in the application architecture (see ❷ in Fig. 5), then the meta-patterns will identify the highlighted elements[5]

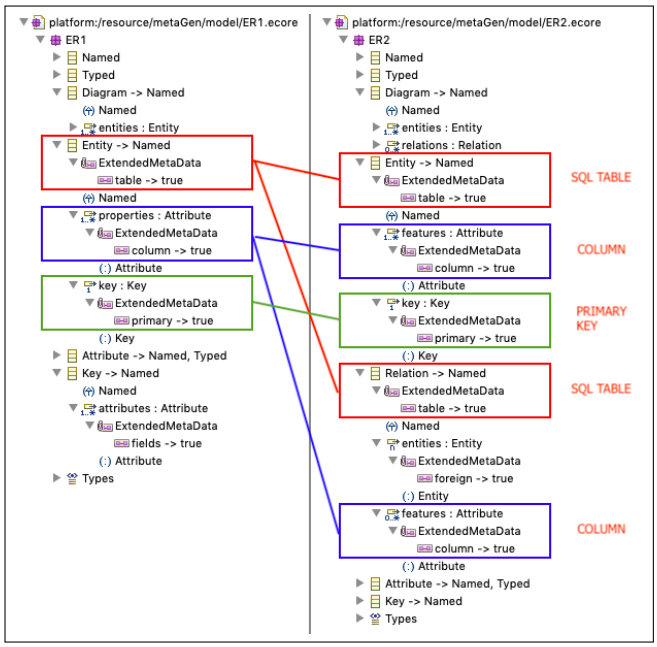

Figure 6 $\texttt{ER}_1$ and $\texttt{ER}_2$ decorated with annotations

A simplified version of the hotello grammar is presented in Fig. 7. The `m-template` non-terminal is the root element denoting an hotello meta-template that consists of an `header` and a list of `commands`. As for the root element, whenever the scope or the context in which the constructs is related to meta- or instance-templates, the corresponding non-terminals are prefixed by `m-` and `i-`, respectively. Thus, the iterators (like those given, e.g., on lines 10 and 12) are specified by `m-iterator` and `i-iterators`. The non-terminal `query` is an EOL[6] expression that predicates on metamodels if occurs in an `m-iterator`, whereas on the specific metaclass and meta-properties in case it occurs within an `i-iterator`.

Once we know how to write a meta-template and a metamodel is given, the hotello translator generates the corresponding instance-template (see ❸) written in EGL. In the case of $\texttt{ER}_1$, the instance-template is given in Listing. 1 in Sect. 2. In particular, the lines 8-10 of the instance-template are generated from the code in lines 10 and 18 in the meta-template once instanciated for the `column` tagged metaclass in $\texttt{ER}_1$, i.e., `Attribute`. Then, the generated instance-template can be applied to any instance of the selected metamodel (❹), e.g., $\texttt{ER}_1$, to produce the target artefact (❺).

As already said, a meta-template purpose is to render model-to-text transformations resilient to metamodel changes. While the changes a metamodel can undergo are arbitrary, their rationale is related to shifting business requirements or new insights emerging from the domain. Therefore, revised metamodels

[5] The annotations have been manually reproduced by the red boxes to enhance the image readability.
[6] https://www.eclipse.org/epsilon/doc/eol/

```
⟨m-template⟩ ::= ⟨header⟩ {⟨command⟩}

    ⟨header⟩ ::= {⟨import⟩}
    ⟨import⟩ ::= ⟨m-imports⟩
               | ⟨i-imports⟩

 ⟨m-imports⟩ ::= 'IMPORT' path ';'
 ⟨i-imports⟩ ::= 'import' path ';'

   ⟨command⟩ ::= ⟨iterator⟩
               | ⟨comment⟩
               | ⟨text⟩
               | ...

   ⟨comment⟩ ::= '/*' string '*/'
               | '//' string

  ⟨iterator⟩ ::= ⟨m-iterator⟩
               | ⟨i-iterator⟩

⟨m-iterator⟩ ::= 'FORALL' '('
                     id ':' ⟨m-pattern⟩ 'TAGGED' id
                 ')' '{'
                     {⟨command⟩}
                 '}'

⟨i-iterator⟩ ::= 'foreach' '('
                     id 'in' id '.' ⟨i-pattern⟩
                 ')' '{'
                     {⟨command⟩}
                 '}'

   ⟨m-pattern⟩ ::= EOL expression
   ⟨i-pattern⟩ ::= EOL expression

               ...
```

Figure 7 The hotello EBNF grammar definition

are usually within a certain proximity from the version they originated from. Such way such mutations affect model-to-text transformations is mitigated by the hotello tagging and annotation mechanism as they allow to abstract from the concrete metaclasses and structural features used in the meta-template. For instance, the meta-template above can be applied to the ER_2 metamodel in the right-hand side of Fig. 6 thanks to the annotations. In particular, by annotating the metaclass `Relation` with `table` the meta-template can generate a table for each of the entities and relations occurring in the corresponding ER_2 instances. The resulting instance-template for the ER_2 metamodel is given in Listing 3.

The class of metamodels class that a meta-template can handle is defined by the meta and instance patterns occurring in it, i.e., the patterns must be valid navigational expressions for the considered metamodel. Such requirement gives place to a notion of *typing* (Steel & Jézéquel 2007) similar, although less expressive, to the transformation *typing requirements model* in the sense of (Lara et al. 2019). The patterns define the navigational expressions as follows: the meta-patterns predicate over the metamodel (metaclasses and metaproperties), whereas the instance-patterns predicate, in turn, over the specific syntactical connectives defined in the selected metamodel. More in detail, in hotello the `m-patterns` occurring in the `m-iterators` identifies the admissible metamodels, whose instances can, in turn, be queries by means of the `i-patterns` present in the instantiated `i-iterators`. In essence, as long as the metamodel mutations do not affect the structures subsumed by the meta- and instance-pattern, the meta-template keeps producing the expected outcome and becoming, therefore, resilient to such changes.

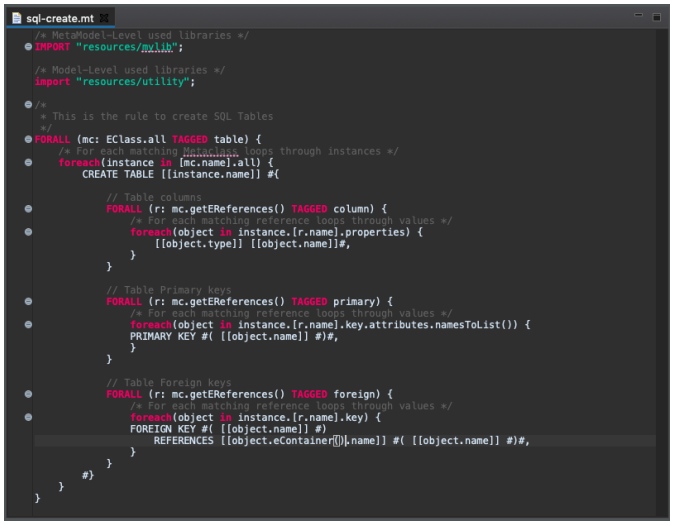

Figure 8 The meta-template edited in the hotello environment

The language has been implemented on the EMF framework using Xtext[7]; from the complete hotello grammar, a parser, a serializer, and a smart editor are generated. Figure 8 shows the meta-template given in Listing 5 while edited in the hotello modeling environment. A multi-step semantic anchoring translates hotello meta-templates into an intermediate EGL-based notation and, in turn, into plain EGL (instance-)templates by means of an Epsilon transformation. For the sake of reproducibility, the complete application alongside the artefacts featured in this paper are publicly available on GitHub[8].

In the next section, a validation of hotello model-to-text transformations is provided in the domain of Internet-of-Things. The objective is to demonstrate how meta-templates can generate component adaptors written in C++ and persist changes performed on the metamodel to accommodate new requirements.

4. Case study: device interfacing in IoT

In this section, we describe a case study realized in the domain of Internet-of-Things (IoT). The domain is characterized by many different devices that typically need to communicate with each other across different protocols and means urging system designers to realize various interface adapters to let parts communicate. Consequently, maintaining such systems becomes critical whenever some device needs to be replaced with a different one no matter the reasons, which results in a considerable

[7] http://eclipse.org/Xtext
[8] https://github.com/MDEGroup/metatemplating

impact on the source code, mostly whether the device uses a different protocol. We show how such an evolutionary scenario can be conveniently managed by employing the hotello approach by automatically creating and updating standard adapters based on designer needs and system requirements. In particular, starting from a domain-specific notation for modeling single isolated IoT systems consisting of many parts that have to communicate by-wire with their central controller, e.g., using protocols like 1-wire[9], I2C[10], or SPI[11]. After that, the initial metamodel is extended to manage systems with remote device communication (e.g., using high-level network protocols like JSON-based or XMI-based) in a System-of-Systems context. In our example, we assume that very low-level functions are provided by vendor libraries, which expose or expect in some way predefined values.

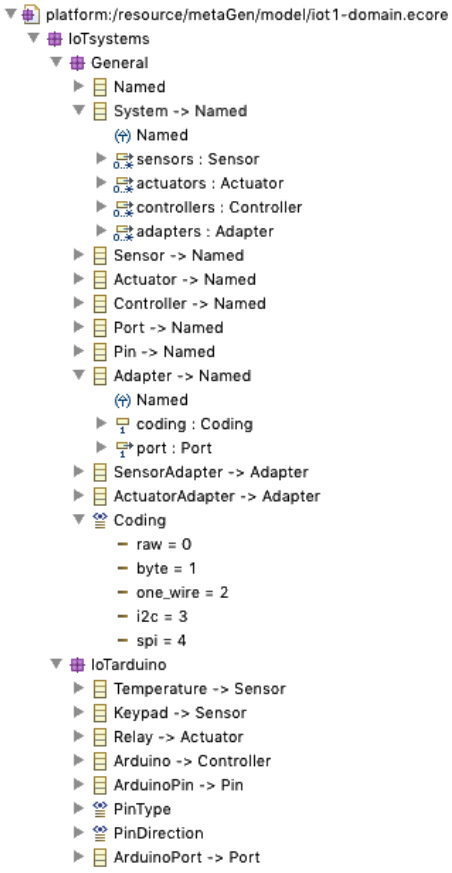

Figure 9 First version of IoT domain metamodel.

Figure 9 describes an initial version $\texttt{MM}_1$ of our IoT metamodel, as intended for designing isolated systems composed of devices connected through `Sensors`, `Actuators`, and `Controllers`. We focus our attention on the `Adapter` concept, the software component that realizes the standardization among peripherals and controllers. We aim at uniforming the values provided by device-specific libraries to a typical object-oriented structure used in controller firmware code. Then, each `Adapter` is linked to a specific `Port` defined onto the controller: such port is a logical grouping of some physical `Pins` used for the electronic interfacing between a controller and a peripheral. All these pins have to be initialized at startup time, so it is suitable to provide an `Initializer` for each controller that setups pins for *input* or *output* at firmware startup.

The rest of the section is organized as follows. First, we show how *adapters* and *pin initializers* can be generated with hotello by:

- describing the corresponding meta-patterns and meta-templates for both categories of components;
- generating the instance-templates, i.e., the model-to-text generators; and
- generating the target artefacts by applying the generators to an $\texttt{MM}_1$ instance.

After that, the metamodel $\texttt{MM}_1$ is extended into $\texttt{MM}_2$ to capture additional requirements. New generators for adapters and pin initializers consistent with $\texttt{MM}_2$ are then consistently obtained while retaining the meta-template unmodified.

4.1. Initial version of the IoT tooling chain

Both adapters and pin initializers are software components that can be automatically generated. According to the process described in Fig. 5, we will define two meta-templates based on the $\texttt{MM}_1$ metamodel by identifying the relevant patterns for each meta-template, the corresponding annotations in the metamodel, and the expected outcome to be generated.

4.1.1. Definition of meta-patterns and templates

In this section, we present the meta-patterns for navigating $\texttt{MM}_1$; the identified patterns for the `Adapters` meta-template are the following:

- **Adapter**: the transformation has to traverse `Adapter` to create the glue-code needed among the system components. An `Adapter` can be distinguished into a `SensorAdapter` or an `ActuatorAdapter` (see Fig. 9) depending on whether it encodes or decodes the data objects exchanges between the controller and the peripheral. It also contains metadata about the producer coding type and communication `Port`;
- **Coding**: an `Adapter` is able to encode/decode data using a set of predefined modes. In this first version, three modes can be used: raw data (directly obtained from the single pins), parallel digital data (obtained from the combination of two or more digital pins), and high-level protocol (provided by the device producer libraries);
- **Port and Pin**: an `Adapter` communicates through logical ports, namely a set of physical controller pins. Each specific type of controller has its characteristic representation for its pins, which at least has to identify the GPIO (General Purpose Input/Output) number and its typology (e.g., analogic, digital, pwm).
- **Naming**: each model element that has to be uniquely identified refers to a common naming system.

[9] https://www.maximintegrated.com/en/products/ibutton-one-wire/one-wire.html
[10] https://www.nxp.com/docs/en/user-guide/UM10204.pdf
[11] https://www.intel.com/content/www/us/en/support/articles/000020952/software/chipset-software.html

Whereas the meta-patterns for the `Pin Initializer` meta-template are:

- **Controller**: each controller defines its used `Pins` as grouped into logical `Ports`. Since there could be non-standard controllers as well, each specific admitted type has to be separately tagged;
- **Pin**: a specific controller has its own definition of `Pin`; for each of them the GPIO number and pin direction (input, output, bidirectional) have to be identified;
- **Naming**: each model element that has to be unambiguously identified refers to a common naming system; in this case, the same naming system for all model elements is used.

The previous lists of patterns require the metamodel to be consistently annotated; each annotation will be grouped under a specific namespace to achieve the separation among meta-templating processes. In detail, `IotHal` has been defined as a namespace for `Adapter` meta-template, and `IotStartup` has been defined as a namespace for `Pin Initializers` one.

We will now describe the definition of the meta-template and the annotation of domain metamodel following identified patterns-of-interest. For the sake of space, we will only describe the process for the automatic creation of `Adapter` generators.

The meta-template for `Adapter` generator is divided into four segments:

(1) a segment of static output, containing some source code that is needed in all cases;
(2) a segment of dynamic output, containing the inclusion of known protocols libraries;
(3) a segment of dynamic output for the construction of reading `Adapters`;
(4) a segment of dynamic output for the construction of writing `Adapters`.

The segment (1) is not affected by any transformation and will appear as it is in the final artefacts. Segment (2) is elaborated in the first step of the meta-generation process and becomes static text for the second step in the instance-template. The corresponding hotello fragment is given in Listing 6.

```
/* Protocols libraries */
FORALL (lib: EEnum.all TAGGED comm_modes) {
  FORALL (proto: lib.eLiterals TAGGED proto) {
  // *.getInfo(⟨TAG⟩) report extra information
  // inserted into the annotation
#include ⟨[proto.getInfo('proto')].h⟩
  }
}

/* Static code - Pin access Interface */
void writeAnalog(int pin, int value);
int readAnalog(int pin);
void writeDigital(int pin, bool value);
bool readDigital(int pin);
/* *** */
```

Listing 6 Meta-template fragment related to segments (1-2)

Segments (3-4) are processed in the first step of the meta-generation process and become dynamic code for the second step when some metamodel-dependent parts are instantiated as a static piece of code. At the same time, the rest expects to elaborate model-dependent information as shown in Listing 7. Because of the duality of segments (3-4), we are discussing segment (3) only.

```
/* Reading Adapters */
FORALL (mc: EClass.all TAGGED in_adapter) {
    foreach(c in [mc.name]) {
        // Get naming attribute from metamodel
        FORALL (mc_name: mc.getAllAttributes() TAGGED naming) {
/* Reading Adapter - [[c.{mc_name.name}]] */
class [[c.{mc_name.name}]]ReadingAdapter {
    public:

    /* Fields */
    FORALL (coding: mc.getAllAttributes() TAGGED coding) {
        FORALL (c_type: coding.eType.eLiterals TAGGED raw) {
            // [...]
        }
        FORALL (c_type: coding.eType.eLiterals TAGGED proto) {
            // [...]
        }
    }
    // [...]
```

Listing 7 Meta-template fragment related to segment (3)

Once the meta-templates are designed, the hotello tooling chain generates the corresponding instance-templates as discussed in the next section.

4.1.2. Generation of instance-templates

The meta-templates defined in the previous section can be actualized with the MM_1 metamodel, i.e., all references to meta-elements occurring in the meta-templates are resolved with actual metaclasses and properties, and the corresponding instance-templates are generated. This automated step is performed by an Epsilon transformation yielding the target EGL code. At this point, `Adapters` and `Pin Initializers` can be synthesized from any model conforming to the MM_1 metamodel.

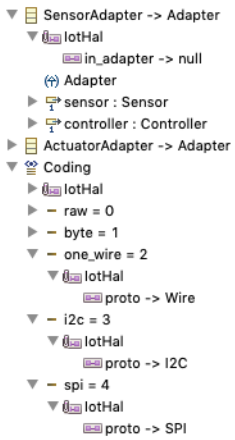

Figure 10 Excerpt of the initial annotated domain metamodel

Figure 10 shows an excerpt of the MM_1 decorated with the annotations corresponding to the `Adapter` meta-patterns, i.e., each different annotation will be used in the guard of the meta-rule to be triggered. In particular, the `SensorAdapter` metaclass is annotated as `inadapter` and some `Coding` literals are annotated as `proto`. These annotations are used in previous defined segment (3) as part of reading `Adapters`.

```
[*-Protocols libraries*]
```

```
#include "Wire.h"
#include "I2C.h"

[*-Adapter for SensorAdapter*]
[%for (c in SensorAdapter) {%]

/* Reading Adapter - [%::=c.name%] */
class [%::=c.name%]ReadingAdapter {
  public:

    [*-Fields*]
  [%if (c.coding.literal ::=:= "raw") {%]
    [*...*]
  [%}%]
  [%if (c.coding.literal ::=:= "byte") {%]
    [*...*]
  [%}%]
  [%if (c.coding.literal ::=:= "one_wire") {%]
    [%for (port in c.port) {%]
  Wire* [%::=port.id%];
      [%for (pin in port.pins) {%]
  const int [%::=port.id%]_[%::=pin.id%] ::= [%::=pin.number%];
      [%}%]
    [%}%]
  [%}%]
  [%if (c.coding.literal ::=:= "i2c") {%]
    [%for (port in c.port) {%]
  I2C* [%::=port.id%];
      [%for (pin in port.pins) {%]
  const int [%::=port.id%]_[%::=pin.id%] ::= [%::=pin.number%];
      [%}%]
    [%}%]
  [%}%]
    [*...*]
```

Listing 8 Excerpt of the instance-template generated from the meta-template

After the metamodel is annotated, the meta-templates in Listings 6 and 7 are translated into instance-templates written in EGL, as shown in Listing 8.

At this point, the final textual artefact can be generate from any $\texttt{MM}_1$ instance as illustrated in the next section.

4.1.3. Generated artefact

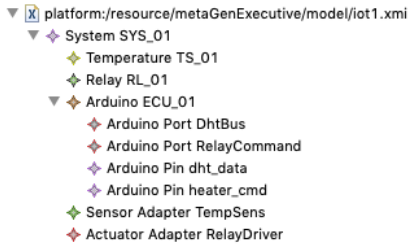

Figure 11 An $\texttt{MM}_1$ instance

Figure 11 presents an examplar of model conforming to the $\texttt{MM}_1$ metamodel. It represents a simple IoT system composed of a single `Arduino` controller with a temperature DHT sensor and a command relay. There are two single pin ports defined on the controller, one for the sensor and another for the relay. DHT sensors use a one-wire digital protocol, which exchanges packet data on a single pin. The device vendor provides its own library to access such information. On the other side, a relay uses a single two-state digital pin to be excited or not. A first `Sensor Adapter` is defined for interfacing the temperature sensor, and it is declared as a 1-wire protocol coding. A second `Actuator Adapter` is given for interfacing the relay, and it is declared as a raw data coding.

Applying the instance-template generated in the previous section to the model in Fig. 11 will produce the C++ code reported in Listing 9.

```
#include "Wire.h"
#include "I2C.h"

// Static code - Pin access Interface

void writeAnalog(int pin, int value);
int readAnalog(int pin);
void writeDigital(int pin, bool value);
bool readDigital(int pin);

// ***

/* Reading Adapter - TempSens */
class TempSensReadingAdapter {
  public:

    Wire* DhtBus;

    const int DhtBus_dht_data ::= 3;

  void run() {

    DhtBus->read();
  }

};

/* Writing Adapter - RelayDriver */
class RelayDriverWritingAdapter {
  public:

    bool heater_cmd;

  void run() {

    writeDigital(2, heater_cmd);

  }

};
```

Listing 9 Result of generation process

4.2. Evolving the IoT metamodel

As already described, a metamodel is a *living* entity that, similarly to software, is prone to changes for the most disparate reasons. Modifying a metamodel does come with the price of adapting the rest of the modeling ecosystem, including models, editors, and transformations, to respond to the changes. In order to illustrate the resilience of the hotello meta-templates, we modify the IoT metamodel $\texttt{MM}_1$ to cover also remote communication management. To this end, the metamodel is extended with new concepts and new kinds of `Adapters`.

The revised metamodel $\texttt{MM}_2$ is presented in Fig. 14. Before discussing the details of the performed evolution, we briefly discuss the two simple (parallel independent) modifications represented in Fig. 12 and Fig. 13, respectively.

In the first modification, the metaclass `Named` is renamed into `UUID` (alonside the attribute `name` that is renamed into `id`). Such change let any instance-pattern using either `Named` or `name` become invalid. However, the meta-template copes with such renaming thanks to the `naming` annotation presents in line 5 in Listing 7 (that is then used in line 7). Therefore, as long as the `UUID` metaclass preserves the `naming` annotation given to `Named`

the meta-template behaves as expected. The annotations can be considered as denoting an equivalence class of concepts that are ontologically similar.

Figure 12 Case 1: model differences between MM_1 and MM_2

The second case in Fig. 13 is another atomic modification, i.e., the addition of the literal enumeration `spi` for denoting a new high-level protocol. It is worth noting that the new literal does not introduce a new kind of knowledge and can be associated with an existing instance-pattern using the right annotation. Indeed, the `spi` literal does not add any information for the scope of `Adapters` generation, and can be treated as `i2c` and `one − wire`. In this case, they are all annotated with `proto`, which specifies that the `Adapter` uniformly refers to a device vendor library that manages the specific protocol. Like this, the instance-template is automatically updated to extend its management to the `spi` coding.

Figure 13 Case 2: model differences between MM_1 and MM_2

Despite the renaming and the literal addition are simple refactorings that should not pose any difficulty, their impact has repercussions on the tooling chain and possibly throughout the modeling ecosystem. It is also worth noting that such changes likely do not completely respond to shifting requirements or domain issues that typically imply considerations on the modified modeling notation's semantics.

As already said, the metamodel MM_2 introduces a set of new concepts that enhances the modeling capabilities by covering aspects that were not considered by the previous metamodel version. The main modifications in the evolution from MM_1 to MM_2 and the rationale motivating them is summarized in Table 1. Interestingly, the motivations are strictly related to the need to extending the formalization of the IoT domain and to overcome the difficulties due to the lack of expressiveness in MM_1.

For instance, the necessity of extending the expressiveness by modeling multiple systems, letting them interact, and provide a specific kind of network adapters is significant. It implies that modelers reported difficulties using modeling notation as they could not fully express their needs. Thus, the modifications extended the notation intended semantics and provided the modelers with additional constructs and connectives for expressing uncovered scenarios. In detail, the new `NetworkAdapter` metaclass is a refinement of `SoftwareAdapter` that manipulates internal data object formats to different string-based types, e.g.,

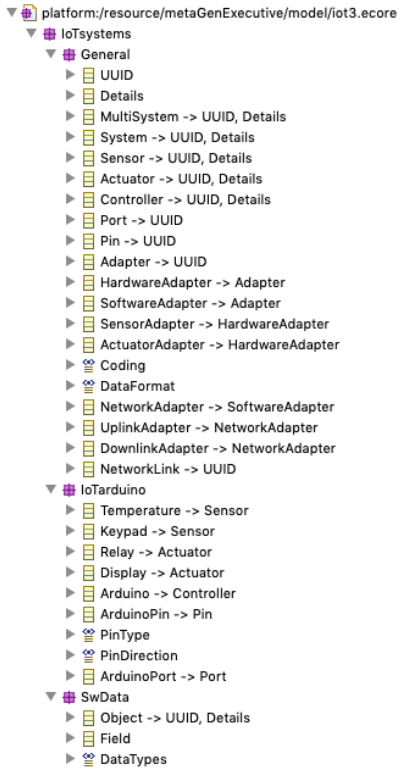

Figure 14 `MM2`, the evolved IoT metamodel

JSON and XML. Such an extension is backward compatible, i.e., existing individual MM_1 conforming models can be imported into a single MM_2 multi-system model.

Despite the semantical relevance of the evolution operated over the modeling notation, the meta-template remains valid, i.e., the instance-templates can then be seamlessly upgraded to MM_2 to produce a more expressive final artefact. Nevertheless, when changes cannot be associated with existing meta-patterns, i.e., the extensions are related to domain aspects left uncovered, they are just ignored by the meta-template leading to information loss.

In the rest of the section, we present the upgrade of the `Adapter`-related instance-template from the same meta-template

requirement	**refactoring**
multiple systems must be considered within the same model	the `MultiSystem` metaclass is added
the expressiveness of the modeling notation must be extended in order to be able to specify also network adapters	the `NetworkAdapter` metaclass is added alongside the metaclasses `DownlinkAdapter`, `UplinkAdapted`, `HardwareAdapter` and `SoftwareAdapter`; in addition, the required connectives (e.g., references in `NetworkAdapter`) have been added
the network adapters within the provided system can interact with each other	the `NetworkLink` metaclasses is added and the `MultiSystem` metaclass is a container for it

Table 1 The main evolution steps from MM_1 to MM_2

given for MM_1 and, in turn, the generation of the final artefact.

4.2.1. Automated upgrade of the instance-template

The instance-template given in Listing 8 is not aware of the differences between MM_1 and MM_2. However, it can be automatically upgraded to MM_2 by simply applying the meta-template to the MM_2 metamodel. The new version of the `Adapter` instance-template given in Listing 10 features code fragments and new EGL statements that were not considered in the previous version.

```
[*-Adapter for SensorAdapter*]
[%for (c in SensorAdapter) {%]

/* Reading Adapter - [%:=c.id%] */
class [%:=c.id%]ReadingAdapter {
  public:

    [*-Fields*]
    [%if (c.coding.literal ::=:= "raw") {%]
      [*...*]
    [%}%]
    [%if (c.coding.literal ::=:= "byte") {%]
      [*...*]
    [%}%]
    [%if (c.coding.literal ::=:= "one_wire") {%]
      [*...*]
    [%}%]
    [%if (c.coding.literal ::=:= "i2c") {%]
      [*...*]
    [%}%]
    [%if (c.coding.literal ::=:= "spi") {%]
      [%for (port in c.port) {%]
    SPI* [%:=port.id%];
        [%for (pin in port.pins) {%]

    const int [%:=port.id%]_[%:=pin.id%] ::= [%:=pin.number%];
        [%}%]
      [%}%]
    [%}%]

    [*...*]
};
[%}%]

[*-Adapter for DownlinkAdapter*]
[%for (c in DownlinkAdapter) {%]

/* Reading Adapter - [%:=c.id%] */
class [%:=c.id%]ReadingAdapter {
  public:

    [*-Fields*]
    [%if (c.coding.literal ::=:= "json") {%]
      [%for (port in c.port) {%]
    JsonFormatter* [%:=port.id%];
        [%for (pin in port.pins) {%]

    const int [%:=port.id%]_[%:=pin.id%] ::= [%:=pin.number%];
        [%}%]
      [%}%]
    [%}%]
    [%if (c.coding.literal ::=:= "xml") {%]
      [%for (port in c.port) {%]
    XmlConnector* [%:=port.id%];
        [%for (pin in port.pins) {%]

    const int [%:=port.id%]_[%:=pin.id%] ::= [%:=pin.number%];
        [%}%]
      [%}%]
    [%}%]

  void run() {
    [*-Reading*]
    [%if (c.coding.literal ::=:= "json") {%]
      [%for (port in c.port) {%]
    [%:=port.id%]->read();
      [%}%]
    [%}%]
    [%if (c.coding.literal ::=:= "xml") {%]
      [%for (port in c.port) {%]
    [%:=port.id%]->read();
      [%}%]
    [%}%]
  }
};
[%}%]
```

Listing 10 Excerpt of the instance-template upgraded by the meta-template

In particular, the instance-template from line 36 on generates the C++ code for network adapters.

4.2.2. Upgraded artefacts

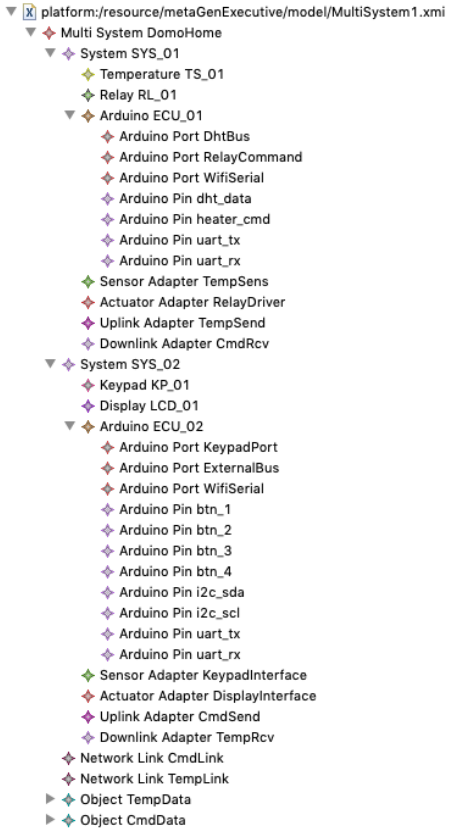

Figure 15 Example model conform to second metamodel version

Figure 15 presents a MM_2 instance describing a model containing two extended IoT systems. It represents the extension of the previous system model in a system-of-systems environment, where systems can interact through network links. In this scenario, it is expected that the generator create adapters for all systems by properly navigating the new model structure, by using the new naming feature and managing all by-wire protocols, including `spi`.

```
#include "Wire.h"
#include "I2C.h"
#include "SPI.h"

#include "JsonFormatter.h"
#include "XmlConnector.h"

// Static code - Pin access Interface
void writeAnalog(int pin, int value);
```

```
int readAnalog(int pin);
void writeDigital(int pin, bool value);
bool readDigital(int pin);
// ***

/* Reading Adapter - TempSens */
class TempSensReadingAdapter {
  public:
    Wire* DhtBus;
    const int DhtBus_dht_data ::= 3;

  void run() {
    DhtBus->read();
  }
};

/* Reading Adapter - KeypadInterface */
class KeypadInterfaceReadingAdapter {
  public:
    bool btn_1;
    bool btn_2;
    bool btn_3;
    bool btn_4;

  void run() {
    btn_1 ::= readDigital(2);
    btn_2 ::= readDigital(3);
    btn_3 ::= readDigital(4);
    btn_4 ::= readDigital(5);

  }
};

/* Reading Adapter - CmdRcv */
class CmdRcvReadingAdapter {
  public:
    JsonFormatter* WifiSerial;

    const int WifiSerial_uart_tx ::= 0;
    const int WifiSerial_uart_rx ::= 1;

  void run() {
    WifiSerial->read();
  }
};

/* Reading Adapter - TempRcv */
class TempRcvReadingAdapter {
  public:
    JsonFormatter* WifiSerial;

    const int WifiSerial_uart_tx ::= 0;
    const int WifiSerial_uart_rx ::= 1;

  void run() {
    WifiSerial->read();
  }
};

/* Writing Adapter - RelayDriver */
class RelayDriverWritingAdapter {
  public:
    bool heater_cmd;

  void run() {
    writeDigital(2, heater_cmd);
  }
};

/* Writing Adapter - DisplayInterface */
class DisplayInterfaceWritingAdapter {
  public:
    I2C* ExternalBus;

    const int ExternalBus_i2c_sda ::= 6;
    const int ExternalBus_i2c_scl ::= 7;

  void run() {
    ExternalBus->write();
  }
};

/* Writing Adapter - TempSend */
class TempSendWritingAdapter {
  public:
    JsonFormatter* WifiSerial;

    const int WifiSerial_uart_tx ::= 0;
    const int WifiSerial_uart_rx ::= 1;

  void run() {
    WifiSerial->write();
  }
};

/* Writing Adapter - CmdSend */
class CmdSendWritingAdapter {
  public:
    JsonFormatter* WifiSerial;

    const int WifiSerial_uart_tx ::= 0;
    const int WifiSerial_uart_rx ::= 1;

  void run() {
    WifiSerial->write();
  }
};
```

Listing 11 Result of generation process

Listing 11 presents the outcome of applying the upgraded instance-template to the model in Fig. 15. The listing contains the C++ code related to one of the two modeled systems, the other code fragment is analogous and has been omitted. As expected, it contains all the needed hardware and software adapters, and a proper C++ `include` directive to support JSON and XML network protocols, and the `spi` by-wire protocol too.

4.3. Discussion

In this section, we have presented an application of hotello in the IoT domain. The objective is to demonstrate that under certain conditions, the meta-templates are resilient to metamodel changes. The conditions are mainly related to the degree of disruptiveness of the modifications performed during the evolutions. On the one side, the rationale behind the metamodel changes depends on the insight coming from the domain, which naturally characterizes the requested changes' deepness. On the other hand, the changes are characterized by their impact on the meta-template's validity, which also has a linguistic or syntactic nature.

It is worth considering that for a better comprehension of the nature of changes, a classification of the impact of metamodel changes can be useful. Following the classification originally introduced in (Gruschko et al. 2007) for the co-evolution of models, metamodel changes can be classified according to the way they affect the instance-template as follows:

- **non-breaking and partial**: changes that do not break the template validity but that include additions of elements that are ignored by the template in the generation process;
- **breaking and resolvable**: changes that breaks the template but that can be resolved deterministically by an automated procedure, e.g., a metaclass or an attribute renaming;
- **breaking and non-resolvable**: changes that break the template validity whose structure and navigation expressions cannot be automatically adapted.

Even though the first category of changes is non-breaking, they reduce the metamodel coverage offered by the instance-templates with a consequent loss of information. Re-establishing the full metamodel coverage typically requires

extending the instance-template with additional rules and navigational expressions, a task that might present limited scalability and is prone to errors. The case study presented above demonstrated that this class of changes can be quickly addressed with hotello. Also, the *breaking and resolvable* changes can be easily resolved, but this is true also for other approaches, e.g., (Di Rocco et al. 2014). Finally, the *breaking and non-resolvable* changes are the most challenging as they might require additional knowledge to be provided by the modeler.

In the illustrated case study, the code related to the interaction between the two systems and modeled employing the `NetworkLink` concept cannot be generated by the meta-template. Such a task would require knowledge not available to the meta-template and cannot be predicted in advance without additional means such as a domain ontology. In other words, this is a case of *breaking and partial* change that borders (if not overlaps) the *non-resolvable* case. Indeed, its solution requires the analysis and understanding the new domain insight and the corresponding requirement elicitation. Nevertheless, the degree of automation introduced by hotello in the upgrade and adaptation of the instance-templates is relevant and represents the central contribution of the hotello approach.

5. Related work

As already discussed, modifying a metamodel in response to shifting business requirements is commonplace in model-driven engineering. Existing approaches typically refer to the problem of restoring conforming or consistency relation between a changing metamodel and different categories of artefacts, including models, model transformations, and model-to-text transformations. As to existing template-based languages, such as Acceleo and EGL, they are single-order mechanism for code/text generation. Hotello is implemented on top of EGL and the instance templates are written in EGL. In this respect, it is difficult to provide a direct comparison between Hotello and them.

Metamodel/model co-evolution. The co-evolution between metamodels and models has been largely investigated over the last decade and more, as witnessed by the survey in (Hebig et al. 2016). The approaches are distinguished in state-based and operation-based (Koegel et al. 2010). The former ones compare two (meta)models regardless of how they are obtained by using, e.g., similarity-based differencing. In (Cicchetti et al. 2008), migration procedures are generated, via higher-order transformations, directly from the metamodel differences; a similar approach is given in (Garcés et al. 2009), where also the equivalences and differences between any pair of metamodels are computed. Other approaches assume that differences are given, and only focus on the generation of the adaptation procedures (Wachsmuth 2007; Herrmannsdoerfer et al. 2008; Vermolen & Visser 2008). In contrast with state-based approaches that tend to be more declarative, operation-based ones can be considered programmatic since they rely on specialized languages, like Flock (Rose et al. 2010) or EMF Migrate (Iovino 2012), or refactoring catalogs (Herrmannsdoerfer 2010). It is worth noting that all available approaches, but EMF Migrate, are specialized on the co-evolution of models only and cannot be applied to the adaptation of other kind of artefacts.

Metamodel/model transformation co-evolution. Not too many approaches are available for the co-evolution of artefacts other than models. One of the main obstacles is related to the strong coupling between the structural definition of a (source) metamodel and the rule-based decomposition of associated transformations (Kurtev et al. 2006). In (Levendovszky et al. 2009), higher-order transformations are used to adapt transformations developed in the GME/GReAT toolset[12]. Another approach is proposed in (García et al. 2012), where atomic metamodel changes are detected and, for each of them, an associated co-evolution is automatically derived. Both approaches automate only parts of the adaptation, thus leaving the missing parts to the modeler's responsibility.

Metamodel/model-to-text transformation co-evolution. Again, not too many approaches are available. In (Di Rocco et al. 2014), an approach to the coupled evolution of metamodels and template-based transformations is proposed. The solution is based on the Acceleo template-based language and is proposed to adapt corrupted templates by means of an ATL (Jouault et al. 2006) transformation that takes the metamodel changes and a model-based representation of the corrupted template, and it returns the adapted transformation.

In addition to the existing approaches, an analysis (Khelladi et al. 2017) proved how the existing approaches for adapting model-to-model and model-to-text transformations could offer only limited support. The approach that we have presented in this paper is entirely different from the existing ones, as it does not rely on the possibility of adapting the template-based generators. Instead, it proposes a new notation that, at the limited price of some additional annotation, makes the artefacts resilient to changes that so far have been considered non-resolvable, as long as the target notation remains unchanged.

6. Conclusions and future work

This paper presented a novel approach based on higher-order techniques that leverage the abstraction in template-based languages. A notion of resilience has been introduced that can make templates persistent to metamodel changes in evolutionary processes. In practice, the meta-template approach defined by hotello exploits the fact that the metamodels involved in the evolution are ontologically related. The approach has been validated on a case study in the IoT domain, thus showing how ranges of modifications can be addressed with little or no effort. Future work regards the investigation of meta-template foundational aspects, and how to increase the expressiveness alongside the needed automation. In particular, we are interested in defining a metamodel-typing notion characterized by meta- and instance-patterns in the meta-templates, because it would increase the overall degree of support and automation. Moreover, we are interested in adopting an annotation mechanism based on domain ontologies, rather than a set of labels, to exploit the structural knowledge encoded in ontologies for validation and assurance purposes. To this end, we will consider

[12] https://www.isis.vanderbilt.edu/Projects/gme/

existing metamodel matching algorithms, such as the work in (Addazi et al. 2016), to detect traceability links identifying the metaclasses that in the current approach are denoted by the same labels. In this respect, it may be interesting to investigate the adoption of metamodel clustering techniques like those in (Basciani et al. 2016), and (Babur et al. 2016), for defining a notion of *proximity* that characterizes the class of metamodels the same meta-template can be applied to. Future work comprises also empirical experiments for validating the approach against existing template-based languages and co-evolution techniques for code-generators.

Acknowledgments

The authors are supported by ERMES (Envisioning Railways systems through Model-driven Engineering approacheS), a project funded by Rete Ferroviaria Italiana (RFI).

References

Addazi, L., Cicchetti, A., Di Rocco, J., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2016). Semantic-based model matching with emfcompare. In *Me@ models* (pp. 40–49).

Babur, Ö., Cleophas, L., & van den Brand, M. (2016). Hierarchical clustering of metamodels for comparative analysis and visualization. In *European conference on modelling foundations and applications* (pp. 3–18).

Basciani, F., Di Rocco, J., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2016). Automated clustering of metamodel repositories. In *International conference on advanced information systems engineering* (pp. 342–358).

Cicchetti, A., Di Ruscio, D., Eramo, R., & Pierantonio, A. (2008). Automating co-evolution in model-driven engineering. In *2008 12th international ieee enterprise distributed object computing conference* (pp. 222–231).

Di Rocco, J., Di Ruscio, D., Iovino, L., & Pierantonio, A. (2014). Dealing with the coupled evolution of metamodels and model-to-text transformations. In *Me@ models* (pp. 22–31).

Di Rocco, J., Iovino, L., & Pierantonio, A. (2012). Bridging state-based differencing and co-evolution. In *Proceedings of the 6th international workshop on models and evolution* (pp. 15–20).

Di Ruscio, D., Iovino, L., & Pierantonio, A. (2012). Evolutionary togetherness: how to manage coupled evolution in metamodeling ecosystems. In *International conference on graph transformation* (pp. 20–37).

Di Ruscio, D., Iovino, L., & Pierantonio, A. (2013). Managing the coupled evolution of metamodels and textual concrete syntax specifications. In *2013 39th euromicro conference on software engineering and advanced applications* (pp. 114–121).

Di Ruscio, D., Lämmel, R., & Pierantonio, A. (2010). Automated co-evolution of gmf editor models. In *International conference on software language engineering* (pp. 143–162).

Garcés, K., Jouault, F., Cointe, P., & Bézivin, J. (2009). Managing model adaptation by precise detection of metamodel changes. In *European conference on model driven architecture-foundations and applications* (pp. 34–49).

García, J., Diaz, O., & Azanza, M. (2012). Model transformation co-evolution: A semi-automatic approach. In *International conference on software language engineering* (pp. 144–163).

Gruschko, B., Kolovos, D., & Paige, R. (2007). Towards synchronizing models with evolving metamodels. In *Proceedings of the international workshop on model-driven software evolution* (p. 3).

Hebig, R., Khelladi, D. E., & Bendraou, R. (2016). Approaches to co-evolution of metamodels and models: A survey. *IEEE Transactions on Software Engineering*, *43*(5), 396–414.

Herrmannsdoerfer, M. (2010). Cope–a workbench for the coupled evolution of metamodels and models. In *International conference on software language engineering* (pp. 286–295).

Herrmannsdoerfer, M., Benz, S., & Juergens, E. (2008). Automatability of coupled evolution of metamodels and models in practice. In *International conference on model driven engineering languages and systems* (pp. 645–659).

Iovino, L. (2012). *Coupled coevolution in metamodeling ecosystems* (Unpublished doctoral dissertation). Università degli Studi dell'Aquila.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., & Valduriez, P. (2006). Atl: a qvt-like transformation language. In *Companion to the 21st acm sigplan symposium on object-oriented programming systems, languages, and applications* (pp. 719–720).

Khelladi, D. E., Rodriguez, H. H., Kretschmer, R., & Egyed, A. (2017). An exploratory experiment on metamodel-transformation co-evolution. In *2017 24th asia-pacific software engineering conference (apsec)* (pp. 576–581).

Koegel, M., Herrmannsdoerfer, M., Li, Y., Helming, J., & David, J. (2010). Comparing state-and operation-based change tracking on models. In *2010 14th ieee international enterprise distributed object computing conference* (pp. 163–172).

Kolovos, D. S., Di Ruscio, D., Pierantonio, A., & Paige, R. F. (2009). Different models for model matching: An analysis of approaches to support model differencing. In *2009 icse workshop on comparison and versioning of software models* (pp. 1–6).

Kurtev, I., Van Den Berg, K., & Jouault, F. (2006). Evaluation of rule-based modularization in model transformation languages illustrated with atl. In *Proceedings of the 2006 acm symposium on applied computing* (pp. 1202–1209).

Kusel, A., Etzlstorfer, J., Kapsammer, E., Retschitzegger, W., Schwinger, W., & Schönböck, J. (2015). Consistent co-evolution of models and transformations. In *2015 acm/ieee 18th international conference on model driven engineering languages and systems (models)* (pp. 116–125).

Lara, J. D., Guerra, E., Ruscio, D. D., Rocco, J. D., Cuadrado, J. S. n., Iovino, L., & Pierantonio, A. (2019). Automated reuse of model transformations through typing requirements models. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, *28*(4), 1–62.

Lehman, M. M., & Belady, L. A. (1985). *Program evolution: processes of software change*. Academic Press Professional, Inc.

Levendovszky, T., Balasubramanian, D., Narayanan, A., & Kar-

sai, G. (2009). A novel approach to semi-automated evolution of dsml model transformation. In *International conference on software language engineering* (pp. 23–41).
Ohst, D., Welle, M., & Kelter, U. (2003). Differences between versions of uml diagrams. In *Proceedings of the 9th european software engineering conference held jointly with 11th acm sigsoft international symposium on foundations of software engineering* (pp. 227–236).
Rose, L. M., Kolovos, D. S., Paige, R. F., & Polack, F. A. (2010). Model migration with epsilon flock. In *International conference on theory and practice of model transformations* (pp. 184–198).
Rutle, A., Iovino, L., König, H., & Diskin, Z. (2020). A query-retyping approach to model transformation co-evolution. *Software and Systems Modeling*, *19*, 1107–1138.
Schmidt, D. C. (2006). Model-driven engineering. *Computer-IEEE Computer Society-*, *39*(2), 25.
Steel, J., & Jézéquel, J.-M. (2007). On model typing. *Software & Systems Modeling*, *6*(4), 401–413.
Van Der Straeten, R., Mens, T., & Van Baelen, S. (2008). Challenges in model-driven software engineering. In *International conference on model driven engineering languages and systems* (pp. 35–47).
Vermolen, S., & Visser, E. (2008). Heterogeneous coupled evolution of software languages. In *International conference on model driven engineering languages and systems* (pp. 630–644).
Visser, E., Warmer, J., Van Deursen, A., & Van Deursen, A. (2007). Model-driven software evolution: A research agenda. *In Proc. Int. Ws on Model-Driven Software Evolution held with the ECSMR*, *7*, 33.
Wachsmuth, G. (2007). Metamodel adaptation and model co-adaptation. In *European conference on object-oriented programming* (pp. 600–624).
Wimmer, M., Kusel, A., Schönböck, J., Retschitzegger, W., Schwinger, W., & Kappel, G. (2010). On using inplace transformations for model co-evolution. In *Proc. 2nd int. workshop model transformation with atl* (Vol. 711, pp. 65–78).

About the authors

Vittorio Cortellessa Vittorio Cortellessa is Professor at the Università degli Studi dell'Aquila (Italy). His research interests are in software performance engineering, software reliability engineering, non-functional software properties, and model-driven engineering. He has published more than 120 articles in international journals and conferences, he has been and currently is part of the organizing committees of several international conferences, including ICSE, ASE, ICSA and ICPE, and he is in the editorial board of Empirical Software Engineering. You can contact the author at vittorio.cortellessa@univaq.it or visit http://people.disim.univaq.it/cortelle/.

Alfonso Pierantonio is professor at the Università degli Studi dell'Aquila (Italy). His interests are in software engineering and, in particular, model-driven and language engineering with special attention to co-evolution techniques, consistency management, and bidirectionality. He has published more than 140 articles in scientific journals and conferences and has been on the organizing committee of several international conferences, including MoDELS and STAF. Alfonso is Editor-in-Chief of the Journal of Object Technology and in the editorial and advisory board of Software and System Modeling, and Science of Computer Programming, respectively. He is co-principal investigator of several research and industrial projects. You can contact the author at alfonso.pierantonio@univaq.it or visit http://pieranton.io.

Tiziano Lombardi is PhD student at the Università degli Studi dell'Aquila (Italy) and registered IT Engineer. His research fields include Model-Driven Engineering and code generation. You can contact the author at tiziano.lombardi@graduate.univaq.it.