Université de Montréal

Evolution of Domain-Specific Languages Depending on External Libraries

par

Khady FALL

Département d'informatique et de recherche opérationnelle (DIRO)
Faculté des arts et des sciences

Mémoire présenté en vue de l'obtention du grade de
Maître ès sciences (M.Sc.)
en Informatique

Orientation Génie logiciel

November 5, 2020

Université de Montréal

Faculté des arts et des sciences

Ce mémoire intitulé

Evolution of Domain-Specific Languages Depending on External Libraries

présenté par

Khady FALL

a été évalué par un jury composé des personnes suivantes :

Michalis Famelis
(président-rapporteur)

Eugène Syriani
(directeur de recherche)

Abdelhakim Hafid
(membre du jury)

Résumé

L'ingénierie dirigée par les modèles est une approche qui s'appuie sur l'abstraction pour exprimer davantage les concepts du domaine. Ainsi, les ingénieurs logiciels développent des langages dédiés (LD) qui encapsulent la structure, les contraintes et le comportement du domaine. Comme tout logiciel, les LDs évoluent régulièrement. Cette évolution peut se produire lorsque l'un de ses composants ou le domaine évolue. L'évolution du domaine ainsi que l'évolution des composants du LD et l'impact de cette évolution sur ceux-ci ont été largement étudiés. Cependant, un LD peut également dépendre sur d'éléments externes qui ne sont pas modélisées. Par conséquent, l'évolution de ces dépendances externes affecte le LD et ses composants. Actuellement, les ingénieurs logiciels doivent évoluer le LD manuellement lorsque les dépendances externes évoluent. Dans ce mémoire, nous nous concentrons sur l'évolution des librairies externes. Plus spécifiquement, le but de cette thèse est d'aider les ingénieurs logiciels dans la tâche d'évolution. À cette fin, nous proposons une approche qui intègre automatiquement les changements des librairies externes dans le LD. De plus, nous offrons un LD qui supporte l'évolution des librairies Arduino. Nous évaluons également notre approche en faisant évoluer un éditeur de modélisation interactif qui dépend d'un LD. Cette étude nous permet de montrer la faisabilité et l'utilité de notre approche.

Mots-clés: Ingénierie dirigée par les modèles, langage dédié, évolution automatique, librairies externes

Abstract

Model-driven engineering (MDE) is an approach that relies on abstraction to further express domain concepts. Hence, language engineers develop domain-specific languages (DSLs) that encapsulates the domain structure, constraints, and behavior. Like any software, DSLs evolve regularly. This evolution can occur when one of its components or the domain evolves. The domain evolution as well as the evolution of DSL components and the impact of such evolution on them has been widely investigated. However, a DSL may also rely on external dependencies that are not modeled. As a result, the evolution of these external dependencies affects the DSL and its components. This evolution problem has yet to be addressed. Currently, language engineers must manually evolve the DSL when the external dependencies evolve. In this thesis, we focus on the evolution of external libraries. More specifically, our goal is to assist language engineers in the task of evolution. To this end, we propose an approach that automatically integrates the changes of the external libraries into the DSL. In addition, we offer a DSL that supports the evolution of the Arduino libraries. We also evaluate our approach by evolving an interactive modeling editor that depends on a DSL. This study allows us to demonstrate the feasibility and usefulness of our approach.

Keywords: Model-driven engineering, domain-specific language, automatic evolution, external libraries

Contents

List of figures

List of acronyms and abbreviations

API	Application Programming Interface
CIM	Computation-Independent Model
COM	Communication port
DSL	Domain-Specific Language
EEPROM	Electrically-Erasable Programmable Read-Only Memory
EGL	Epsilon Generation Language
EMF	Eclipse Modeling Framework
I/O	Input/Output
I2C	Inter-Integrated Circuit
LED	Light-Emitting Diode

M2M	Model-to-model transformation
M2T	Model-to-text transformation
MDA	Model-Driven Architecture
MDE	Model-Driven Engineering
OCL	Object Constraint Language
PIM	Platform-Independent Model
PCB	Printed Circuit Board
PSM	Platform-Specific Model
PWM	Pulse Width Modulation
RFID	Radio-Frequency IDentification
SE	Software Engineering
TBCG	Template-Based Code Generation
UML	Unified Modeling Language

Acknowledgements

I would like to take this opportunity to express my sincere gratitude and my appreciation to my supervisor Prof. Eugène Syriani. He was very present during the last two years of my bachelor and throughout my master. He has been an invaluable support during my anxious periods and has been very understanding and patient with me.

I would also like to take this opportunity to thank my colleagues at the GEODES lab for their help, their advice and their support. In particular, I would like to thank Vasco Sousa who never hesitated to help me and share his great knowledge.

I dedicate this thesis to my parents, brothers and sister. I have been blessed with their wise advice, unwavering support and prayers throughout my life.

Chapter 1

Introduction

1.1. Context

Software engineers have always sought to improve productivity during software development by improving abstraction. Therefore, many approaches have been proposed throughout the years. Model-driven engineering (MDE) [**74**] is one of such approaches. It promotes models and transformations (which themselves can be represented as models) as first class entities. MDE technologies combine domain-specific languages (DSLs), transformations engines, and generators. DSLs [**89**] often represent an abstraction of a combination of function invocation at the generated code level in a syntax and semantics closer to the domain than to the code. Transformation engines (e.g., [**42, 44**]) are used to generate models from a source language to a target language while generators (e.g., [**27, 73**]) are used to generate source code from models.

Like any software, systems built using model-driven approaches need to evolve. In particular, since the specifications and requirements of domains evolve over time, DSLs are bound to evolve. Therefore, many evolution and co-evolution scenarios of DSL components have been investigated [**59**]. As a result, many metamodel-model co-evolution approaches have been proposed [**37**]. As metamodels and models are not the only components likely to change, research has also been done on the evolution of other DSL components such as the concrete syntax of the language [**20, 64**], the constraints or the static semantics [**9, 16**], and the transformations [**19, 54**].

1.2. Problem statement and thesis proposition

In the literature, the evolution of a DSL is only tied to the evolution of its components (typically those components that rely on the metamodel). However, a DSL is a restricted modeling language. Hence, it often relies on dependencies that are not explicitly modeled, such as compilers, libraries, and debuggers. Therefore, their evolution can also affect the

DSL. In particular, many DSLs nowadays rely on external libraries which are in constant evolution. For example, suppose a DSL is used to specify the coordination of micro-services and that a new version of one micro-service is available. If the new version only provides refactoring to improve the performance and has no impact on the concepts it represents, then the DSL itself will not be impacted. If some of its remote functions changed signature, the code generator of the DSL would need to be updated accordingly. Another situation is if a new micro-service is available providing functionalities for a new concept, then the DSL needs to evolve to offer this concept to the user, and the code generator would also need to evolve accordingly. Thus, the evolution of an external library may affect the syntax and the semantics of a DSL.

Evolving a DSL with non-modeled artifacts has yet to be considered [**65**]. In current DSL evolution practices [**59**], the language engineer must evolve manually the metamodel, rebuild, and regenerate all DSL components whenever external libraries are updated or new ones appear. There are four main problems we foresee in this practice.

- New errors may be introduced when manually modifying the metamodel or concrete syntax. For example, adding the new concept to the class diagram of the metamodel may require extensive refactoring of the metamodel, instead of simply adding a sub-class at the right location.
- Inconsistencies may occur between the new metamodel and associated artifacts, such as editors, code generators, test suites or model transformations that must take into account the new meta-concepts and make the correct function invocations to the external library.
- This may also lead to incompatibilities between the new DSL and models defined with the previous version of the DSL.
- The language becomes static and inflexible since representing library functions in a metamodel is like hardwiring function calls in the syntax of the language.

Therefore, automating the integration of the new functionalities and concepts is a possible solution to reduce the manual changes to the current structure of the DSL and its associated components.

To this end, we propose, in this thesis, a process that minimizes the language engineer's manual effort of evolving the DSL when a library it depends on evolves. This process is completely transparent to the DSL user. As functionalities available in an external library are independent of the core of the DSL, we opt to separate the language in two parts: the core and the extension. The core language regroups the stable concepts, with respect to evolution of the language, while the extension language is composed of the functionalities from external libraries. Our process does not modify the syntax nor the semantics of the language manually constructed by the language engineer, but only extends it.

1.3. Contributions

The goal of this thesis is to help language engineers cope with the evolution of external libraries on which DSLs depend on by automatically integrating the changes in the DSLs. The contributions of this thesis are the following:

- An approach to evolve DSLs automatically when changes occur in the external libraries they depend on in three key steps: specifying the extension library, extending the DSL syntax, and extending the DSL semantics.
- ArduinoDSL, a modeling language to generate Arduino configurations and programs from high-level specifications.
- A case study automatically evolving ArduinoDSL when new Arduino devices and/or new Arduino libraries are available.

1.4. Outline

This thesis is organized as follows. In Chapter 2, we introduce modeling editors and present existing works on language evolution. We also present the Arduino environment. In Chapter 3, we present ArduinoDSL, our first contribution. In Chapter 4, we depict the overall architecture of our evolutionary process. The implementation of our approach is detailed in Chapter 5. In Chapter 6, we show the feasibility, usefulness, and applicability of our approach based on a case study. Finally, we conclude in Chapter 7.

Chapter 2

Background and state of the art

In this chapter, we review different notions of modeling languages. We also discuss related works on modeling language evolution and on application programming interface (API) evolution.

2.1. Modeling editors

From a practical point of view, MDE is a generative process. The most used tools that are generated are the modeling editors. They rely on a modeling language specification and allow users to manipulate models. Modeling editors are an integral part of language workbenches [**31**] which are the set of tools needed to define, reuse, and combine languages and their editors.

2.1.1. Domain-specific languages

A DSL is a modeling language defined for a specific type of problems called domain. Thus, the user is a domain expert who defines domain-specific models using the concepts and notations of the DSL. A DSL is composed of three main components: an abstract syntax, a concrete syntax, and a semantics.

2.1.1.1. **Abstract syntax**. The abstract syntax of a DSL defines the concepts of the language and their relations. Additionally, the abstract syntax may define constraints of the domain. An abstract syntax is usually specified by a metamodel using UML class diagrams notation [**58**]. Therefore, the metamodel is mainly composed of classes, attributes, and associations. Classes represent the concepts of the domain and the types of the model elements. The properties of the concepts are encapsulated in the attributes of the classes. The relations between the concepts are expressed using associations.

The metamodel restricts the allowed types in valid instances. It is possible to further constraint models by defining static semantics typically expressed as Object Constraint Language (OCL) constraints [**68**]. For example, constraints may specify that the value of the name attribute of a class must be unique across all its instances. Many tools exist to describe metamodels, such as Emfatic [**14**].

```
1 package drawing;
2 class Toolbar {
3   val Shape[*] shapes;
4   val Pencil[*] pencils;
5 }
6 abstract class Shape {
7   unique attr String name;
8   ref Pencil drawingTool;
9 }
10 class Rectangle extends Shape {
11   attr int width;
12   attr int height;
13 }
14 class Pencil {
15   attr String lead;
16 }
```

Listing 2.1. Definition in Emfatic of the Toolbar, Shape, Rectangle, and Pencil classes inside the package drawing

Listing 2.1 shows the definition of the classes `Toolbar`, `Shape`, `Rectangle`, and `Pencil` in Emfatic. A `Toolbar` contains multiple `Shapes` and `Pencils`. To express the containment relation between classes, the `val` keyword is used. The `Shape` class is abstract which means that it is not instantiable. It has one attribute which is its name. The name attribute should be unique to the `Shape` which is why the keyword `unique` is used. As to draw a shape, a pencil is needed. Thus, there is an association from `Shape` to `Pencil`. This simple association is expressed using the keyword `ref`. The `Rectangle` class has two attributes that defines its width and height. Since a rectangle is a type of shape, `Rectangle` inherits from `Shape`. The inheritance link is represented by the `extends` keyword. As the width and height cannot be negative, we add some static semantics using the Epsilon Validation Language (EVL) [**52**] in Listing 2.2. The constraint `PositiveLengths` checks that the attributes `rows` and `columns` of every instance of the class `Rectangle` have positive values. If not, the message on line 4 in will be displayed.

```
1 context Rectangle {
2   constraint PositiveLengths {
3     check : self.width >= 0 and self.height >= 0
4     message : "The width and the height cannot be negative."
```

```
5   }
6 }
```

Listing 2.2. Constraints defined in EVL for the class diagram in Listing 2.1

There exist other textual tools to describe class diagrams like TCD [**92**] which allows to describe class diagrams with ASCII [**40**] text, PlantUML [**71**] which supports the description of class diagrams directly within the source code of the software with specialized comments. Tools such as AToMPM [**83**], Visual Paradigm (VP)[1], and GMF [**33**] allow users to describe graphical metamodels using a combination of shapes, symbols, and text.

2.1.1.2. **Concrete syntax**. The concrete syntax of a language is the representation of the abstract syntax of the language. Typically, the concrete syntax of a DSL is either textual or graphical. A textual concrete syntax is commonly defined by grammars [**11, 26, 43**].

```
1  grammar org.xtext.example.xtextexample.XtextExample with org.eclipse.xtext.common.Terminals
2  generate xtextExample "http://www.xtext.org/example/xtextexample/XtextExample"
3  Toolbar:
4    'Toolbar' '{'
5      ('pencils' '{'pencils+=Pencil ( "," pencils+=Pencil)* '}' )?
6      ('shapes' '{' shapes+=Shape ( "," shapes+=Shape)* '}' )?
7    '}';
8  Shape: Rectangle;
9  Pencil: 'Pencil' 'with' 'lead' name=STRING;
10 Rectangle:
11   'Rectangle' name=STRING '{'
12     ('width' width=INT)?
13     ('height' height=INT)?
14     ('drawingTool' drawingTool=[Pencil|STRING])?
15   '}';
```

Listing 2.3. Textual concrete syntax of the Toolbar, Shape, Rectangle, and Pencil classes using Xtext

Listing 2.3 is a grammar developed in Xtext to represent the metamodel in Listing 2.1. The containment reference is expressed using the notation « *variable+=ClassName* » as seen on line 3. Classes are represented by production rules. The attribute *name* is defined in the subclasses. The variable *name* is also a keyword in Xtext. Using it ensures the unicity of the attribute. Moreover, it allows easy cross-referencing which is why it is used instead of `lead` as seen on line 9. The association relation is expressed by referencing the associated class as seen on line 13. This grammar allows to specify the textual concrete syntax of the DSL.

When the concrete syntax is graphical, it takes form of shapes, symbols, and texts. The concrete syntax should be unambiguous. To help researchers and designers, the authors in

[1]Visual Paradigm: `https://www.visual-paradigm.com/`

[**61**] provide a set of principles for visual notations. There are many tools (e.g., GMF [**1**], EuGENia [**51**], Sirius [**88**], Graphiti [**2**], AToMPM [**83**]) available that support graphical notation for the concrete syntax.

```
1 @namespace(uri="drawing", prefix="drawing")
2 @gmf
3 package drawing;
4 @gmf.diagram
5 class Toolbar {
6   @gmf.compartment
7   val Shape[*] shapes;
8   @gmf.compartment
9   val Pencil pencil;
10 }
11 abstract class Shape {
12 @gmf.label(label.pattern="name = {0}")
13   unique attr String name;
14   @gmf.link(width="2")
15   ref Pencil drawingTool;
16 }
17 @gmf.node(figure="rectangle", color="0,128,255", size="150,100", label.placement="none")
18 class Rectangle extends Shape {
19   @gmf.label(label.pattern="width = {0}")
20   attr int width;
21   @gmf.label(label.pattern="height = {0}")
22   attr int height;
23 }
24 @gmf.node(figure="svg", svg.uri="platform:/plugin/org.eugenia.example.eugeniaexample/svg/pencil.
       svg", size="150,150", label.placement="none")
25 class Pencil {
26   @gmf.label(label.pattern="lead = {0}")
27   attr String lead;
28 }
```

Listing 2.4. Graphical concrete syntax of the Toolbar, Shape, Rectangle, and Pencil classes using EuGENia

Listing 2.4 is the annotated version of Listing 2.1 using EuGENia. On line 2, we apply the annotation `@gmf` to the package to mention that GMF-related annotations are to be expected in the package. The root of the metamodel is specified using the annotation `@gmf.diagram`. As a `Toolbar` instance contains `Shape` (resp. `Pencil`) instances, we denote this containment by creating compartments in the `Toolbar` instance which will contain `Shape` (resp. `Pencil`) instances. The `Shape` class does not have a concrete representation since it is an abstract class. However, its attribute `name`, which will be inherited by the class `Rectangle`, has a

concrete representation. It is represented as a text. The association between `Shape` and `Pencil` is represented by a solid line with a width of 2 pixels. The source of the association is an instance of `Shape` and the target is an instance of `Pencil`. The concrete syntax of the `Rectangle` class is a blue rectangular shape with a width of 150 pixels and a height of 100 pixels. Both of its attributes are represented as texts. The `Pencil` class is represented by a Scalable Vector Graphics (SVG) [**10**] image. With the EuGENia annotations, a graphical concrete syntax has been defined for this DSL.

2.1.1.3. **Semantics**. The semantics of a language [**35**] is expressed by using a semantic mapping to link the abstract syntax of the language to a semantic domain. The semantic domain is a well-defined formalism with well-defined semantics. The semantic mapping associates an element from the language's syntax to its meaning in the domain. Every element of the language must have exactly one meaning. The semantics of a DSL is typically defined by operational or translational semantics. Operational semantics defines the behavior of the language. In operational semantics, computations are explicitly modeled. Translational semantics is a type of semantics where the meaning of the language's elements are given in terms of another language. In this thesis, we focus on translational semantics defined by means of a code generator.

2.1.2. Code generation

Code generation refers to the process by which a model is transformed into source code using a code generator. The MDE approach promotes the abstraction of the systems to help domain experts to model using domain terms. In 2001, the Object Management Group launched the Model-Driven Architecture (MDA) initiative [**78**] which is a framework for standardizing MDE. At the top level of abstraction, we have the computation-independent model (CIM) which is transformed to a platform-independent model (PIM) through a model-to-model transformation (M2M). Using another M2M, the PIM is transformed into a platform-specific model (PSM). Finally, the PSM is converted to source code with the help of a code generator.

There exists many code generation approaches, such as code annotations and template-based. A tool that uses the code annotations approach allows its users to add elements using annotations or comments in the source code or in the metamodel. Examples of such approach are JavaDoc [**53**] which generates documentation from annotations, C++ attributes [**76**] which provide additional information to the compiler, or Doxygen [**86**] which generates documentation from comment blocks. Template-based code generation (TBCG) uses templates which describe the structure of the target source code. Templates have two parts: a static part and a dynamic part. The static part is composed of fragments of the source code that will be produced in the output file. The dynamic part acts like a placeholder. It

contains some meta-code that evaluates and selects the different elements to be put in the placeholder. There are several available tools [**82**] using the TBCG approach such, as Xtend [**27**], Xpand [**28**], Acceleo [**62**] and the Epsilon Generation Language (EGL) [**73**].

```
1 [% operation Board!Rectangle create_rectangle() {%]
2  Rectangle [%self.name.first()%] = new Rectangle([%self.width.first()%],[%self.height.first()%]
       );
3 [% } %]
```

Listing 2.5. A template in EGL to instantiate a Rectangle object from the metamodel in Listing 2.1 in Java

In Listing 2.5, we have an excerpt of a template in EGL to instantiate `Rectangle` objects. The template mixes static and dynamic contents. On line 1, we create a new function in EGL that will be called for every instance of the `Rectangle` class in the `Board` model. On line 2, we get the name, width, and height of the object by using a dynamic placeholder.

2.1.3. Editor generation

As the use of DSLs became more prominent, a need to have tools supporting and simplifying their creation and reuse arised. A model editor allows DSL users to create and edit domain-specific models. Additionally, some tools support the combination of multiple DSLs, M2M transformations and code generation. The model editor is generated from the abstract and concrete syntax. The generation process also produces other artifacts related to the editor, such as an analyzer that tokenizes the elements, a parser that exposes the underlying structure in a form of an abstract syntax tree, a serializer that converts the model into a persistent format, an API that describes the language, code generators to transform the model in a target language, etc.

Most editor generators are either textual or graphical.

Textual editor generators, such as Xtext [**11**], Rascal [**85**], Spoofax [**46**] and SugarJ [**29**] are the most popular type as they are not limited by the available technologies. Most of the times, the domain model is defined by means of grammars and programming languages. From the grammar or source code defined by the user, the editor generator derives a full-fledged editor including a parser. The parser can complicate the composition of languages as, for example, composition of two LR parsers [**50**] does not necessarily result in a valid LR parser.

Figure 2.1 shows the textual editor generated from Listing 2.3 using Xtext. The editor is composed of a text area. The reserved words of the grammar are highlighted in purple. Xtext provides also a content-assist as seen on Figure 2.1. The content-assist combined with the cross-referencing allow us to choose one of the `Pencil` instances previously created.

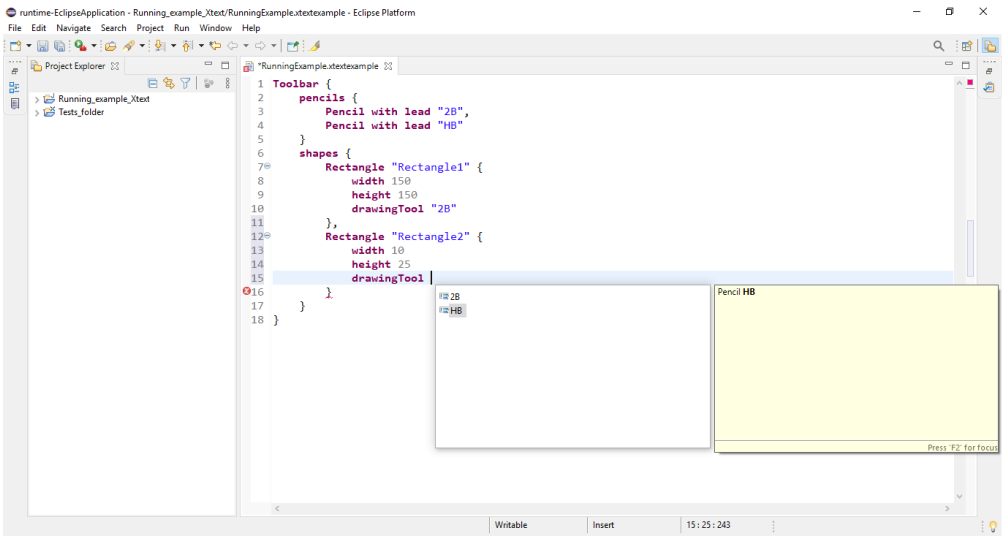

Fig. 2.1. The textual editor generated from Listing 2.3 using Xtext

Many graphical tools were developed (MetaEdit+ [**77**], AToMPM [**83**], Eugenia [**51**], GMF [**33**]) to allow users to manipulate graphical representations of models. Several tools allow the user to describe the domain model using a metamodel. The graphical definition can be specified by adding annotations in the metamodel, drawing the different shapes or by defining a metamodel. Depending on the graphical tool, the user may also need to define a mapping model between the graphical definition and the domain model and a tooling model to be able to use the different shapes. From the models defined, the editor generator will produce the components of the graphical editor. Similarly to textual tools, the generator will produce an API, a static analyzer, etc. Additionally, the graphical editor generator will produce components for the graphical definition, such as a tool palette and a canvas. Since a same shape can be used across different DSLs for different elements, language composition may lead to ambiguity. Figure 2.2 shows the generated editor from Listing 2.4 using EuGENia. On the far right, there is the tool palette with all the model elements. The canvas is where the model elements will be placed. Three elements were added: a Pencil and two Rectangles. The Pencil has a lead of type 2B. The two Rectangles, named *Rectangle 1* and *Rectangle 2*, use the 2B Pencil as a drawing tool. The shapes and leads created on the canvas are strongly typed by the metamodel. Both the textual and graphical editor generators produce editors that are conform to the metamodel and the static semantics.

2.2. Modeling language evolution

In MDE, the metamodel evolution and model co-evolution problem is one of the most researched subject. Many approaches were proposed in the literature mostly based on metamodel differences, traces, or transformation. However, the evolution of a language is not only

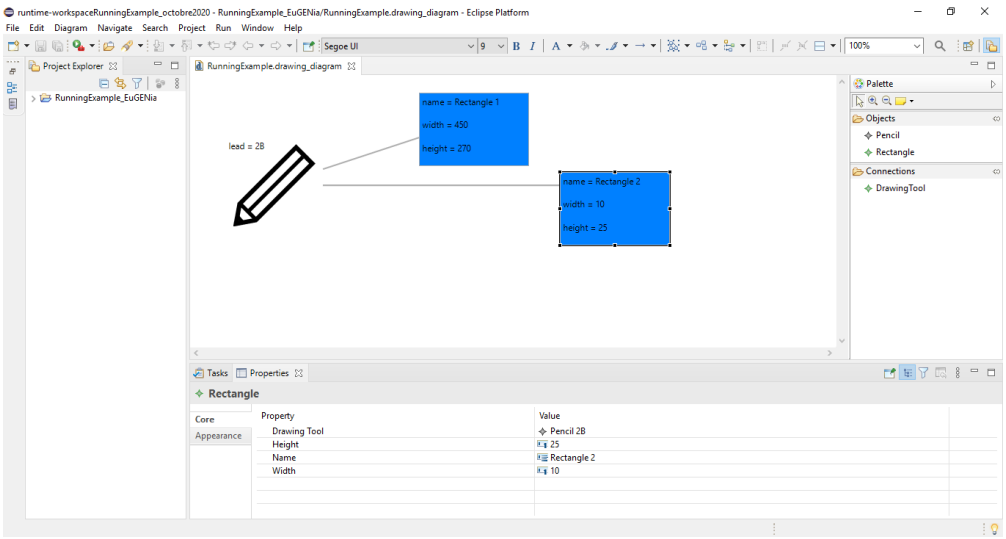

Fig. 2.2. The graphical editor generated from Listing 2.4 using EuGENia

triggered by the evolution of its metamodel. In [**59**], the authors identified four basic scenarios of evolution which can be combined to create new evolution scenarios: model evolution, image evolution, domain evolution, and transformation evolution. A model evolution occurs when a model evolves. Such an evolution does not impact other elements of the language. But, if the model is used as the input of a transformation, the transformation has to be executed again. When the metamodel of a transformation model evolves, the transformation itself has to co-evolve to conform to the new metamodel. A domain evolution is triggered by the evolution of the DSL metamodel. Metamodel evolution impacts the model that has to co-evolve to conform to the new metamodel. Evolving the metamodel also requires evolving all the components that depend on it: the concrete syntax, the constraints, the transformations, the code generators, and the artifacts generated like the modeling editor.

In the following sections, we will look at different approaches proposed in the literature for different language artifacts' evolution.

2.2.1. Model co-evolution

One evolution problem is the metamodel-model co-evolution. The goal is to evolve the models so that they conform to the new version of their metamodel. We can identify three atomic changes that are applied to elements in metamodel evolution: addition, deletion, and modification. These atomic changes can be aggregated to compose a complex change. The approaches proposed in [**49, 87**] use difference models to detect and reconstruct complex evolution traces. The changes applied to a metamodel can be classified in three groups: non-breaking changes, breaking and resolvable changes, and breaking and unresolvable changes [**13**]. Non-breaking changes are changes that do not affect the conformance of models to their metamodels. When there is a non-breaking change, all the models that conformed to the

original version of the metamodel are still compliant to the new version. Breaking changes affect the conformance of the models. If the models can be co-evolved automatically using some migration strategies, the changes are resolvable. However, if no migration strategy can be applied and the language engineer has to manually give additional information, the changes are considered unresolvable.

Many approaches for model co-evolution are proposed in the literature. A classification was proposed in [**37**] based on the resolution strategy: whether they use predefined resolution strategies [**13, 91**], learn from user-specified strategies [**7**], use transformation languages [**63, 72, 80**], generate resolution strategies [**15, 60**], apply constrained-based searches [**18, 47**], or identify complex metamodel changes [**49, 87**]. However, at a higher level, we can distinguish two main types of approaches: comparison approaches and pattern-based approaches.

A comparison approach analyzes the original version of the metamodel and its evolved version, then it compares them. By matching the two (2) versions, it creates a difference model. The difference model is a record of the changes and it will be used to generate migration strategies. Such migration strategy is used in [**13**]. The authors proposed an automatic model co-evolution approach which is based on difference models. The difference model is conform to a difference metamodel derived from the Kernel MetaMetaModel (KM3) [**43**]. The difference metamodel consists of new constructs representing the possible modifications according to a list of possible changes. Two difference models are generated, one for breaking resolvable changes and the other for breaking unresolvable changes. Using each model as input (in any order) of a high-order transformation, the co-evolved model is generated. Similarly to [**13**], the approach proposed in [**32**] use difference models. However, in [**13**] they have a predefined list of possible change while in [**32**] the changes that may occur are not known. Tools such as EMFCompare [**84**] and SiDiff [**75**] are used to generate the difference models.

Pattern-based approaches rely on patterns that map the elements of the old version of the metamodel to elements of its new version. As the metamodel evolves, the changes are recorded to create traces. These traces are then used to generate migration strategies that will be applied to the models so that they can conform to the evolved metamodel. COPE [**39**] is one of the pattern-based approaches proposed in the literature. COPE supports the reuse of migration strategies since it differentiates between metamodel-only changes, metamodel-independent changes, metamodel-specific changes and model-specific changes. When the changes applied to the metamodel do not affect the models, they are classified as metamodel-only changes. In that case, no migration strategy is generated since those changes do not require the migration of the models. Metamodel-independent changes such as renaming a class are changes that can be applied to any metamodel. Therefore, the migration strategies generated from such changes can be reused for any model co-evolution. On the other hand, the migration strategies generated from metamodel-specific changes

cannot be reused as they are only relevant for a given metamodel. When the language engineer has to specify additional information during the migration process, the changes are model-specific. Thus, the migration strategies generated are not reusable. Another patten-based approach is the Model Change Language (MCL) [**63**]. It provides migration rules for the models. A migration rule is composed of a left-hand side (LHS) and a right-hand side (RHS). The LHS consists of an element of the old metamodel while the RHS is an element of the new metamodel. The LHS and the RHS are linked by a *MapsTo* relation which means that the LHS has evolved to the RHS. To specify that an element was previously evolved, a *WasMappedTo* link is added.

For comparison and pattern-based approaches, the conformance of the co-evolved model to the evolved metamodel is verified at the end of the migration process.

2.2.2. Concrete syntax evolution

Few approaches focused on concrete syntax evolution. In [**80**], the authors considered that changes to the concrete syntax do not require domain evolution. They argue that the concrete syntax is used only for display purposes, and carries no semantic information. However, this is not always the case as the concrete syntax can be used to deduce partially or entirely the abstract syntax. Such approach was taken by the authors of [**45, 64, 93**].

In [**93**], the authors proposed an algorithm that uses the existing textual concrete syntax of the domain to induce the abstract syntax. The heuristic conversion process is done by hand and transforms from YACC [**41**] to C++. Therefore, whenever the concrete syntax changes, the domain has to evolve accordingly.

In [**45**], the authors developed a tool designed to simplify the development of the abstract and concrete syntax. It allows users to specify each of them only partially as long as the sum of the fragments allows deduction of the complete syntax. Therefore, the abstract and concrete syntax should be complementary.

In [**64**], the authors proposed an approach to generate rewritable abstract syntaxes from textual concrete syntaxes by means of annotations. They defined six annotations to refine the abstract syntax: *omission* for tokens that can be omitted or elided, *labeling* to distinguish, merge or rename fields or node classes, *boolean access* to indicate that a field should be of type boolean, *list formation* to indicate that the productions for a non-terminal rule describe a list, *inlining* to insert some fields directly into a class, and *superclass formation* to model inheritance.

The authors of [**20**] also addressed the problem of propagating metamodel changes to textual concrete syntax but, unlike the previous approaches, they provided an automated support. They focused on TCS (Textual Concrete Syntax) [**43**] which is a tool for specifying the textual concrete syntaxes of DSLs. They proposed an approach based on model

differencing and model transformations. The approach follows three key steps: (1) identify the dependencies between the metamodel and the concrete syntax definition, (2) classify the metamodel changes according to their consequences over the syntax definition, and (3) define for each category of changes the corresponding adaptations to be operated over the syntax definition in order to restore its consistency with the metamodel.

To the best of our knowledge, only [**21**] focused on graphical concrete syntax evolution. In [**21**], the authors looked at how to adapt GMF editors when changes are applied to the metamodel. The authors argue that manually changing the graphical concrete syntax in GMF is error-prone and labor-intensive. Their approach consists of three elements: (1) *Difference calculation* to identify the changes between two version of a same model, (2) *Difference representation* to represent the previously identified differences in a manipulable way (e.g., a difference model), and (3) *Generation of the adapted GMF models* by inputting the previous difference model to a model-to-model transformation.

2.2.3. Constraint evolution

As metamodels evolve, the artifacts that are related to them have to evolve accordingly. This is also the case of the constraints. Even if constraints are static semantics of a language, their evolution and co-evolution with other DSL components have not been sufficiently investigated.

The Cross-Layer Modeler (XLM) [**16, 17**] is an approach that relies on manually defined template constraints to automatically update the model with the new constraints whenever the metamodel evolves. The constraint templates are composed of a generic part, which contains the common aspects between constraints, and of a variable part that defines the variable points in the template. XLM incrementally evolves the constraints whenever there is a modification in the metamodel. When a modification is detected, the template engine is notified and it uses the information in that notification to determine the actions to adapt the constraints to the new metamodel. XLM defines an evolution action for each of the atomic changes discussed in Section 2.2.1. For an element added in the metamodel, the template engine looks through the constraint templates and instantiate the relevant ones. For a deleted element in the metamodel, the template engine deletes all the constraints related to that element. When an element is updated in the metamodel, the template engine finds all the constraints affected by the modification and replace the outdated values by the new ones.

In [**48**], the authors propose an approach that considers alternative resolutions per impacted part of an OCL constraint and let the user choose which resolution strategies are to be applied. The resolution strategies proposed depend on three key factors: the type of metamodel change, the location of the impacted element in the OCL constraint (in the

context or in the body of the constraint) and the context of the impacted constraint. The changes supported by the approach are at the atomic (add, delete and update) and complex (any combination of the atomic changes) levels. To propose the most suitable resolution strategies in respect to the OCL constraint, the tool first identifies the type of changes that were made. After the user confirms the list of complex changes, an ordered trace of the changes is generated. For each element in the metamodel, a list of the related contraints is generated. Using the trace of changes, the impacted OCL constraints are identified. The influencing factors are determined and resolution strategies are proposed. The atomic and complex changes that do not impact the OCL constraints are ignored.

In [**9**], the authors used a meta-heuristic search based on genetic algorithms for the co-evolution of metamodels and OCL constraints. Their approach, similarly to [**48**], proposes a set of potential candidate solutions to the user who has to manually choose the appropriate evolution. However, instead of identifying the changes of the metamodel that were made, the approach compares two versions of the metamodel and computes a set of the atomic differences. Then, crossover and mutation operations are applied to generate a set of solutions. As the set may be large, a recommendation system based on two strategies is used: a ranking strategy which rank the solutions by using objective functions, and a clustering strategy which produces subsets of similar solutions and choose one solution in each cluster.

2.3. API evolution

In the programming world, programs evolve. This is often reflected by the evolution of their API. An API is a set of methods that allow communication between applications. It also provides documentation on those methods, such as how to create and call them. Often, the terms API and library are used interchangeably. However, even if they are related, they are different. An API only describes the methods while a library is an implementation of those methods. In the rest of this thesis, we use the terms API and library indiscriminately since the approach presented applies to both. We can distinguish two types of APIs: local APIs and web APIs. A local API is an API without network interactions while a web API is accessible through the network.

Many works addressed API evolution and the impact of that evolution on the consumers. In [**55**], the authors identified 16 change patterns that can be divided in two categories: changes that cause compile-time errors and changes that cause run-time errors. The authors found that 80% of API changes are refactorings as they are related to renaming, changing or splitting methods and/or variables. Therefore, another classification [**22, 23**] which also consists in two categories can be identified: changes that are for refactoring and those that are not for refactoring.

As the use of APIs became prominent, tools were developed to help automate their evolution process. CatchUp! [**38**], a plugin for the Eclipse IDE, is one of such tools. It is only focused on refactorings and performs the evolution in two steps: *recording* then *replaying* the changes. First, it records how the library developer changes the API. To do so, a trace of the changes performed by the library developer is collected by the IDE. Then, the trace and the old and new versions of the library are used by the plugin to evolve the client application. The tool generates source code stubs for all the classes in the library then deletes the old version of the library. Finally, it replays the evolution of the library by performing the changes in the client application. All the refactorings are done in the client application by using Eclipse's refactoring objects[2]. Another tool that also automates the evolution of APIs is ReBA [**24**]. Similarly to CatchUp!, it only supports changes that are refactorings and is implemented as an Eclipse plugin. ReBA and CatchUp! share also a similar approach as they both record and replay the changes performed in the library. However, instead of updating all the artifacts to the newer version as CatchUp! does, ReBA creates an adapted-library which will contain all the the APIs that the old client requires. By doing so, ReBA supports both the old and the new version of the library.

Some authors [**30, 70**] focused on the evolution of web APIs which present unique characteristics and new challenges [**55**]. One of the main differences and challenges is the management of deleted methods. The existing approaches for local APIs evolution either support both versions of the library or keep a copy of the deleted method. However, this option is not available to web APIs. After a certain time, the older version of the API will not be provided as a service anymore and the developers have no access to deleted methods. In [**30**], the authors looked at the evolution of Web Services Description Language (WSDL) [**12**] specifications. They developed VTracker which is a tree-differencing algorithm. To be able to analyze WSDL documents, XML representations are produced. The XML documents can then be inspected and compared by VTracker. The authors of [**70**] also look at the evolution of web APIs defined in WSDL. Their tool WSDLDiff extracts and parses two versions of the WSDL interface to create two EMF models, one for each version. By matching the two models, their differences are detected and reported to the user. The differences can take the form of additions, removals, moves and modifications.

2.4. Arduino

Arduino [**8**] started as a research project by Massimo Banzi, David Cuartielles, Tom Igoe, Gianluca Martino, and David Mellis at the Interaction Design Institute of Ivrea (IDII). It is based on two other projects: Processing and Wiring. Wiring and Processing were aimed at non-programmers, more specifically to artists, architects and designers. In 2005, Arduino

[2]Eclipse Refactor Actions: `https://help.eclipse.org/2019-12/topic/org.eclipse.jdt.doc.user/reference/ref-menu-refactor.htm?cp=1_4_6_0`

was forked from Wiring to support the ATmega8 microcontroller[3] which was cheaper than the ATmega168 microcontroller[4] on which Wiring is based. As its precursors, Arduino is aimed at a non-technical audience. It was intended to help design students to create working prototypes connecting the physical world to the digital world.

2.4.1. Arduino board

The Arduino board is an electronic board with a microcontroller. The microcontroller has the advantage of being easily transportable due to its small size, has a low power consumption and an affordable price. The microcontroller is an integrated circuit mainly used in embedded systems. It consists of a microprocessor, data memory, programmable memory and auxiliary resources such as input/output (I/O) ports, converters and timers. The microprocessor will process the information and send instructions. The data memory is a volatile memory for storing temporary data while the programmable memory is a non-volatile memory. It contains the program instructions and is of the EEPROM type, i.e., it can be reprogrammed.

The Arduino board has the advantage of being inexpensive and works on all platforms. In addition, the software and the plans of the board are open-source and extensible.

The board used in the examples in the remainder of this thesis is the Arduino Leonardo board.

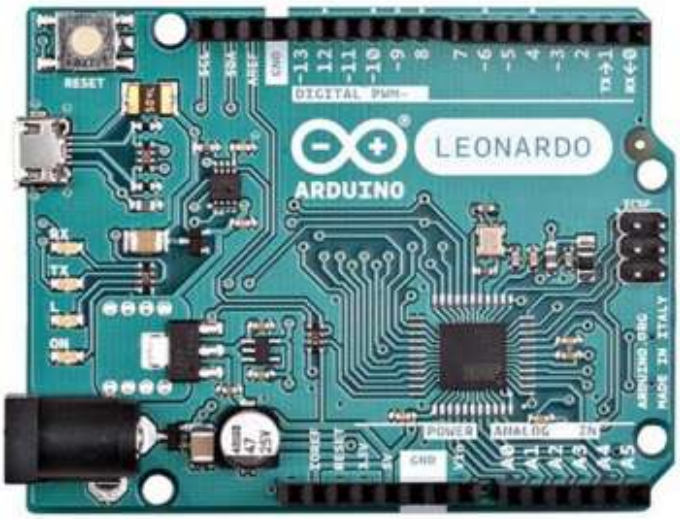

Fig. 2.3. The Arduino Leonardo microcontroller board[5]

[3]ATmega8 microcontroller: `https://www.microchip.com/wwwproducts/en/ATmega8`
[4]ATmega168 microcontroller : `https://www.theengineeringprojects.com/2018/09/introduction-to-atmega168.html`
[5]Image taken from `https://store.arduino.cc/usa/leonardo`

The Arduino Leonardo board[6] is based on an ATMega32u4 clocked at 16 MHz allowing the USB port to be managed by a single processor, which increases the flexibility of communication with the computer. The Arduino Leonardo board can appear on a connected computer in the form of a mouse, a keyboard or a virtual serial/COM port.

I/O pins on the outer edges of the PCB allow a series of add-on devices to be plugged in. There are 20 I/O pins, 12 of which can be used as analog inputs and seven of which can be used as PWM outputs. Pulse width modulation (PWM), is a technique for obtaining analog results with digital means. Digital control is used to create a signal that is processed by components that act as open and closed switches.

The Arduino Leonardo board also has a reset button, a power socket and a USB port.

2.4.2. Arduino IDE

The Arduino IDE[7] is an open-source platform-independent application written in functions from C and C++. It allows the user to write an Arduino code, compile it and upload

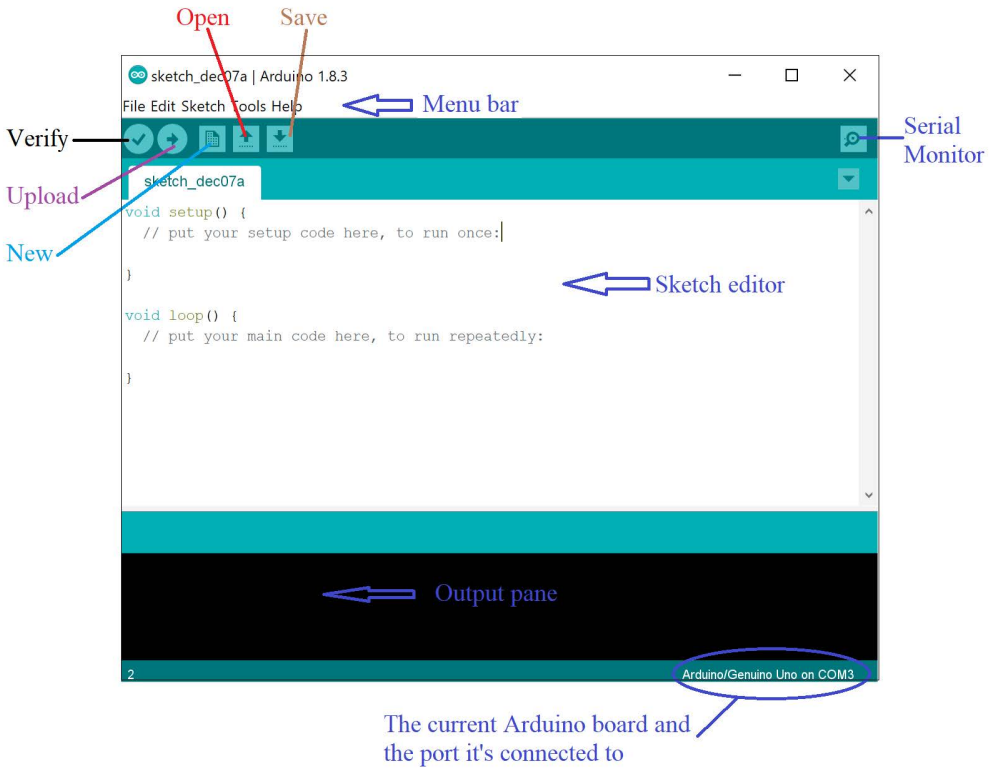

Fig. 2.4. The Arduino IDE

it to an Arduino board.

The Arduino IDE interface can be divided in three main parts: the command buttons, the sketch editor, and the output pane.

[6]Arduino Leonardo: `https://www.arduino.cc/en/Main/Arduino_BoardLeonardo`
[7]Arduino IDE: `https://www.arduino.cc/en/main/software`

2.4.2.1. **The command buttons**. There are six buttons under the menu bar. The check mark appearing in the circular button is used to verify the code. The arrow key will upload and transfer the required code to the Arduino board. The dotted paper is used for creating a new file. The upward arrow is reserved for opening an existing Arduino project. The downward arrow is used to save the current running code. The button appearing on the top right corner is a Serial Monitor. It will open separate pop-up window that acts as an independent terminal and plays a vital role for sending and receiving serial data.

2.4.2.2. **The sketch editor**. In this area, the user writes its Arduino code. Arduino programs are called *sketch* and are written in a simplified C/C++ language [**67**]. An Arduino program has at least two functions: the `setup()` function and the `loop()` function. In the `setup()` function, the user specifies the code to configure the devices and any code that must run once at startup. The `loop()` function will have the code of the behavior that runs in an infinite loop. It is possible to add some user-defined functions.

2.4.2.3. **The output pane**. The message window area will display the memory used by the code and errors occurred in the program while compiling it.

On October 19, 2019, the Arduino Pro IDE[8] was introduced. This new IDE supports Arduino, Javascript, and Python code and has a debugger.

2.4.3. Arduino language

The Arduino Language Reference[9] regroups the structure, the values and the functions of the Arduino programming language. The structure of the Arduino programming language combines the different elements of the Arduino code including control structures and operators. The values of the Arduino programming language are mostly the data types and their conversion functions, and the constants of the language. One of the constants is `LED_BUILTIN` which represent the number of the pin on which the on-board LED is connected. The on-board LED is present in almost all Arduino boards. Arduino pins can be configured either as `INPUT` or `OUTPUT`. The pin configured as INPUT pins are said to be in a *high impedance* state while those configured as OUTPUT are in a *low impedance* state. Pins in a high impedance state only allow a small amount of current through whereas those in a low impedance state provide a larger amount of current to the electrical circuit. More precisely, a pin is configured as an INPUT if it is used to read values from a device, and as an OUTPUT when it is used to write values to a device. Arduino digital pins can only read or write two values: `HIGH` and `LOW`. The meaning of the two values depend if the pin is configured as an INPUT or an OUTPUT. If the pin is configured as an OUTPUT, LOW means the lowest voltage of the

[8]Arduino Pro IDE: `https://www.arduino.cc/pro/arduino-pro-ide`
[9]Arduino Language Reference: `https://www.arduino.cc/reference/en/`

board (0V), and HIGH means the highest voltage of the board (5.5V or 3.3V depending on the board).

In the following, we present the functions at the core of the Arduino language and the declaration of functions.

In its core, the Arduino language is composed of functions that are independent of any device. The functions are used to control the Arduino boards, the devices and to perform computations.

The most used functions are the ones for controlling the input and the output of analog and digital pins: `analogRead()`, `analogWrite()`, `digitalRead()`, `digitalWrite()` and `pinMode()`. The `pinMode()` function is used to configure the pin to be either as INPUT or OUTPUT. The `analogRead()` and `digitalRead()` are used to read the values from a pin, respectively an analog pin and a digital pin. As mentioned before, the value read from a digital pin is either HIGH or LOW. The `analogRead()` function will outup an integer value between 0 and 1023. The functions `analogWrite()` and `digitalWrite()` are used to write a value to a pin. Similarly to `digitalRead()`, `digitalWrite()` can only write HIGH and LOW to a digital pin. The `analogWrite()` function allows integer ranging from 0 to 255 and is not limited to analog pins.

It is possible for the user to create her own functions. There are two ways to create functions: use a function prototype or not. A function prototype is a type of function declaration that does not contain the body of the function. It only has the function's name, return type and the type of its arguments. A function prototype always ends with a semicolon. When a function prototype is used, it has to be declared before the `loop()` function while the usual function declaration containing the body of the function has to be declared after the `loop()` function. If a function prototype is not used, the function is declared before the `loop()` function.

```
1 int add(int, int);
2 int sub(int x, int y) {
3   return (x - y);
4 }
5 void setup(){
6   int sub_result = sub(87,65);
7 }
8 void loop() {
9   int add_result = add(4,987);
10 }
11 int add(int x, int y) {
12   return (x + y);
13 }
```

Listing 2.6. Declaration of the functions `add()` and `sub()`

Listing 2.6 shows an example with the both declaration methods. The `add()` function is declared using a prototype while the `sub()` function is declared directly.

2.4.4. Arduino libraries

In this section, we look through the creation and use of Arduino libraries as well as the standard Arduino libraries since the Arduino language relies heavily on external libraries.

In Arduino, an external library consists of a header file (like in Listing 2.7) and a class file that implements the library.

```
1 class RFID125 {
2   public:
3     void branch(int8_t pin1, int8_t pin2=-1);
4     String readCode();
5     void writeCodes(String code);
6     bool codeIsPresent(String code, bool lowLevel=false);
7 };
```

Listing 2.7. Excerpt of the RFID125 library

Arduino libraries are used to extend the language. They allow the user to create and/or use functions not available in the standard Arduino language. One of the most common libraries used in Arduino is the `Serial` library which is built-in the Arduino framework. It offers functions to communicate with devices and peripherals through serial ports. The Arduino language allows the user to create new libraries (following the structure in Listing 2.7) and integrate them in the Arduino code using the `#include` statement. Arduino libraries are often used to integrate and manipulate new devices. When a new device is developed, it may not be possible to configure it with existing Arduino functions. Therefore, the developer creates a library using C++ to allow the user to perform actions with the device. The user can also create her own libraries to be able, for example, to reuse functions in multiple sketches.

2.4.5. Grove base shield

There are two ways to connect devices to the Arduino board: use a breadboard or use a Grove shield.

The breadboard is a device that does not need soldering. It is used for making and testing the prototype of an electrical circuit. Most electronic components of electrical circuits can be interconnected by inserting their conductors or terminals into the holes in the breadboard, and then making connections through cables if necessary. The holes in the breadboard are connected to each other by metal strips.

To use the breadboard with an Arduino board, you need to have some knowledge of electronics. For example, when connecting devices to the breadboard, it is necessary to think about putting the right resistors to avoid the devices overheating.

Grove[10] makes the connection easier. It is a modular and easy to use system designed to easily connect a processor, such as an Arduino, to a wide range of modules. Unlike the breadboard, using Grove does not require any knowledge of electronics.

The heart of the Grove system is the Base Shield.

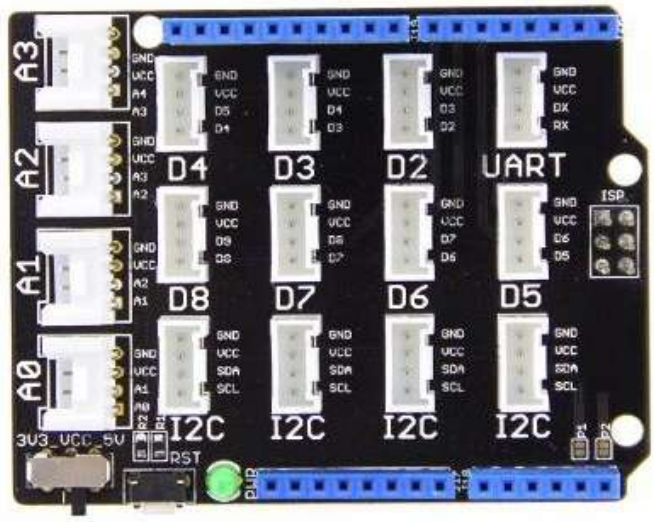

Fig. 2.5. Grove Base Shield v2.0 for Arduino[11]

At the time of writing, the latest version for Arduino is the Grove base shield v2.0[12]. Each module can be connected to the base shield through the appropriate pin which are general purpose input/output (GPIO) ports. The Grove Base Shield includes :

- **Seven digital pins** which can be used for both digital input (e.g., check a button's state) and digital output (e.g., power a light). Digital sensors can only output 0 or 1. However some of the digital pins can function as PWM outputs which allows to get analog results.
- **Four analog pins** which can read the signal from an analog sensor (e.g., temperature sensor) and convert it into a digital value that we can read. Analog sensors can return readings ranging from 0 to 1024.
- **One universal asynchronous receiver-transmitter (UART) pin** which is the Arduino's default port for serial communication with the computer.
- **Four inter-integrated circuit (I2C)** pins which enable communication between devices on a single circuit board.

[10]Grove: `https://www.seeedstudio.com/category/Grove-c-1003.html`
[11]Image taken from `https://wiki.seeedstudio.com/Base_Shield_V2/`
[12]Grove Base Shield v2.0: `https://www.seeedstudio.com/Base-Shield-V2.html`

The Grove Base Shield also integrates a green light emitting diode (LED) to indicate power status, a reset button and a power toggle switch to select the suitable voltage (5V or 3.3V) depending on the microcontroller card used. In the remaining of this thesis, when the context may be potentially ambiguous, we will use the term *device* to refer to *Grove modules* since the term *module* is overloaded in software engineering.

2.4.6. Grove devices

At the time of writing, Grove had developed over 300 different devices[13] that can be grouped into five categories: environmental sensors, motion sensors, wireless devices, user interface devices and physical sensors.

2.4.6.1. **Environmental sensors**. Environmental sensors are used to monitor and report on the environment. Two examples of environmental sensors are light sensors to sense light and air quality sensors to measure air quality.

2.4.6.2. **Motion sensors**. Motion sensors such as the 3-Axis Compass or the 6-Axis Accelerometer&Gyroscope allow the microcontroller to detect motion, location and direction.

2.4.6.3. **Wireless devices**. Wireless devices enable wireless communication ability such as radio-frequency (e.g., Serial RF Pro) and Bluetooth (e.g., Serial Bluetooth).

2.4.6.4. **User interface devices**. User interface devices are used to interact with the microcontroller. They can be input devices such as the Thumb Joystick and the Touch Sensor, or output devices like the LED and the LCD RGB Backlight.

2.4.6.5. **Physical sensors**. Physical sensors are designed to help analyze the physical word. For example, the Ear-clip Heart Rate Sensor is used to measure the heart rate while the PIR (Passive Infrared Sensor) Motion Sensor allows to sense human movement in its range.

2.5. Modeling for Arduino

As the popularity of Arduino grew, many tools were developed to help programmers to write Arduino code. In the following section, we look through some of them.

Arduino Designer[14] is a graphical DSL based on Sirius [**88**], mainly aimed at novice programmers. It only supports modeling for the Arduino DFRduino UNO R3[15]. It only allows for ten (10) predefined devices to be plugged. However, it allows creating custom functions meaning that extension functions can be added.

[13]Grove devices: `https://wiki.seeedstudio.com/Grove/`
[14]Arduino Designer: `https://github.com/mbats/arduino`
[15]DFRduino UNO R3: `https://www.dfrobot.com/product-838.html`

Unlike Arduino Designer, YAKINDU Statecharts Tools (SCT) for Arduino[16] supports many boards thanks to its generic implementation. It is a tool that allows the programmer to develop Arduino code by using Statecharts [**34**]. YAKINDU SCT is based on Eclipse [**33**] and to be able to develop Arduino code, the user has to integrate the Sloeber plugin[17]. Sloeber can also be used on its own. Differently from the Arduino IDE, Sloeber allows to directly manipulate the libraries since any library added to the project, if not found in the source folder, will be downloaded.

Another tool based on Statecharts is QP-Arduino[18]. It is an event-driven framework that was developed to allow Arduino programs to handle multiple events at once. Traditionally, Arduino programs are executed sequentially. Therefore, while waiting for a specific event to occur, Arduino programs are not responsive to any other event. Since QP-Arduino and YAKINDU SCT for Arduino have a generic implementation, the connection between the hardware and the Statecharts model must be added manually in the generated code. Therefore, this can be considered as an alternative solution to integrate external libraries. They do not represent concepts and functions explicitly. The evolution resides in changing the Statecharts model manually, by importing external libraries and invoking the functions of the APIs.

ThingML [**36**] is also a tool based on Statecharts. The ThingML framework is platform-independent. It has two components: a textual modeling editor and a family of code generators. The editor allows its users to model distributed systems using Statecharts. The code generators are in the form of model-to-text transformations and each is aimed at a specific language such as Java, Javascript, and Arduino. When building an Arduino Statechart model, the users write the Arduino code directly in the states. Then, the Arduino code will be added as-is in the generated sketch file. The ThingML framework only supports the evolution of its family of code generators. New functionalities of a language can be integrated by creating a ThingML API (the code in the target language wrapped using ThingML structures) for existing code generators or by using the HEADS[19] code generation framework to create a new code generator. Adding the functionalities of a specific language in the ThingML DSL will defeat the purpose of the framework as it will not be platform-independent anymore. It can only be done by creating sub-languages for each target platform (e.g., ThingML Arduino, ThingML Java) as seen with QP and YAKINDU SCT.

Arduino CLI[20] is a command line application which can be used to configure Arduino boards, compile and upload sketches to them. However, it only creates the skeleton of the sketch files with empty `setup()` and `loop()` functions. The user has to add the Arduino code

[16]YAKINDU Statecharts Tools (SCT) for Arduino: `https://github.com/wendehals/arduino_sct_tools`
[17]Sloeber: `https://github.com/Sloeber/arduino-eclipse-plugin`
[18]QP-Arduino: `http://www.state-machine.com/arduino/`
[19]HEADS: `http://heads-project.eu/`
[20]Arduino CLI: `https://github.com/arduino/arduino-cli`

using another editor. To use external libraries, the user specifies a keyword in Arduino CLI library search command line. Arduino CLI will search through GitHub repositories.

ArduinoDroid[21] is an Arduino IDE for Android. The user can create, compile and upload sketches to Arduino boards. External libraries can be added manually by importing a zipped library or by downloading it.

[21]ArduinoDroid: `https://www.arduinodroid.info/`

Chapter 3

The ArduinoDSL modeling language

In this chapter, we present our first contribution, ArduinoDSL.

3.1. Presentation of the ArduinoDSL language

ArduinoDSL is a DSL for Arduino that runs on the Eclipse Modeling Framework (EMF) [**81**]. The goal of ArduinoDSL is to help novice programmers to write and reason about Arduino code and facilitate the implementation. To this end, ArduinoDSL abstracts away the C++ code as much as possible and generates the code to be deployed on an Arduino board.

ArduinoDSL is a combination of two sub-languages: a conceptual part and a behavioral part. The conceptual aspect of ArduinoDSL is a graphical DSL to model the board configuration and the devices, while the behavioral aspect is a textual DSL used to model the behavior and the device interaction.

3.2. Conceptual aspect

The conceptual part of ArduinoDSL is a graphical DSL that allows users to configure the board and the devices. They can specify which devices will be used and on which pins they will be plugged. Additionally, the user can define the properties of the devices and the libraries that should be used, as well as their default values.

3.2.1. Metamodel

In ArduinoDSL, the conceptual metamodel defines the language for the board part. We depict its class diagram in Figure 3.1. The root of the metamodel is the `Project` class in which we define the project name. This name will be used as the name of the Arduino file when generating the Arduino code. Projects are deployed on a board. At the time of writing, only Arduino boards are supported.

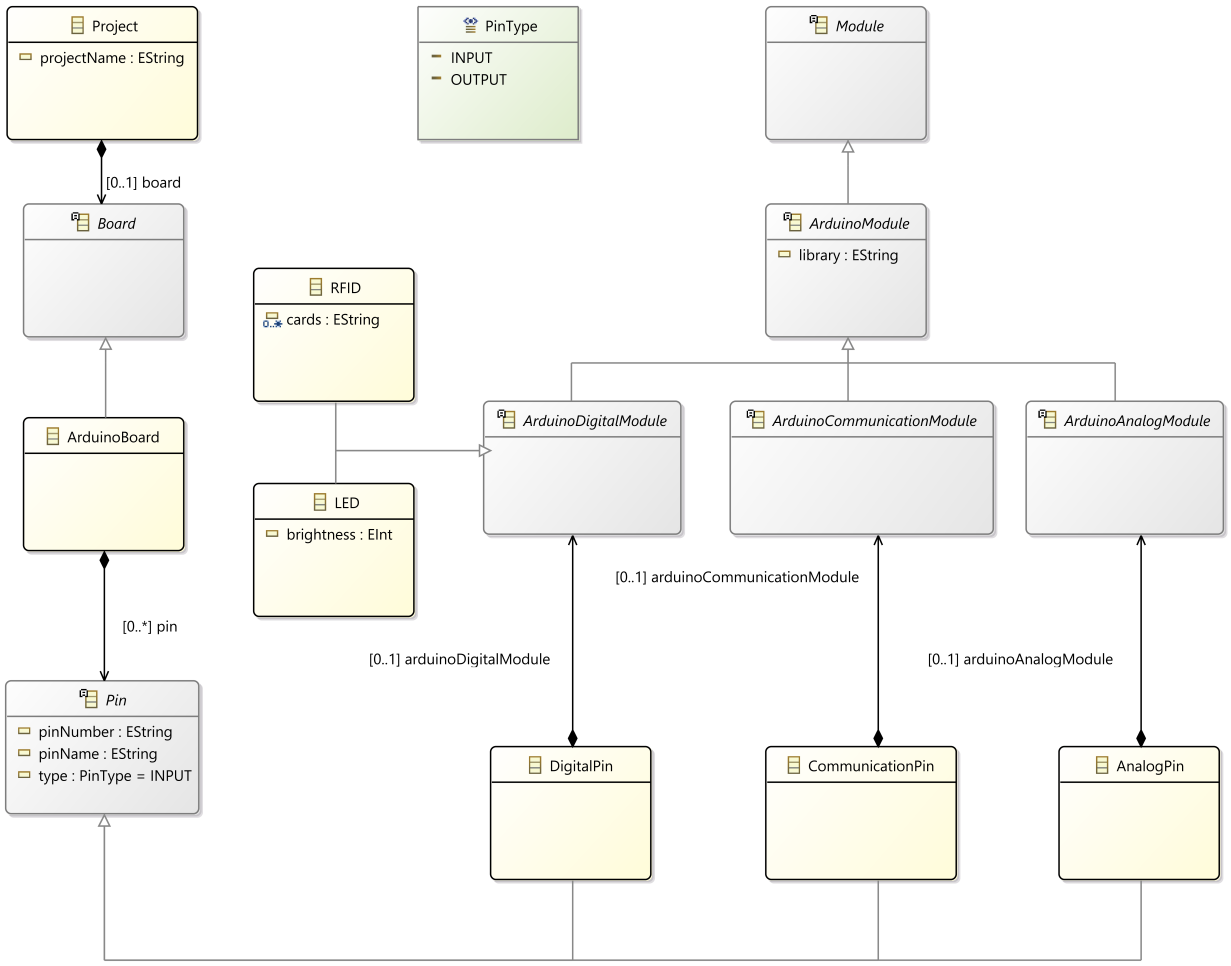

Fig. 3.1. The metamodel of the conceptual aspect of ArduinoDSL

We consider the devices as the main concepts in the conceptual metamodel. Therefore, since the Arduino board is an integral part of an Arduino project, it is represented in the metamodel. From an Arduino board point of view, devices only differ by the pins they are connected to. Hence, instead of dividing the devices in the categories defined in Section 2.4.6, they are grouped according to the pins they can be plugged to. As mentioned in Section 2.4.5, we have four types of pins. However, the UART and the I2C pins are both used for communication. Therefore, we combine them in one category. We then have the three categories of pins and, by extension, three categories of devices: digital, analog, and communication. The `Pin` class has three attributes: `pinNumber`, `pinName`, and `type`. The `pinNumber` attribute is the name or list of names of the physical pins (e.g., *A0*). Instead of referring to a pin by its number, the user can define a user-friendly `pinName`. It can be used to identify the device connected to the pin (e.g., *rfidSensor*). The `type` attribute can be either INPUT or OUTPUT (ref. Section 2.4.3).

We only show the light-emitting diode (LED) and the radio frequency identification (RFID) reader as digital devices in the metamodel since they are going to be used for our examples. An RFID reader is a device to identify and track tags. A tag defines a unique identifier (UID) to attach to objects, such as RFID cards. An LED emits light when current flows through it.

External libraries are used to connect and use almost any device in Arduino (ref. Section 2.4.3). This is why, the `ArduinoModule` class defines a `library` attribute. Its default value is `NONE` which means that the device does not use any external library. When generating the Arduino code, the library will be imported.

We constructed the class diagram implementing this metamodel in an extensible way so that further concepts can be added later on. For instance, following SOLID principles [**57**], the pin classes contain abstract classes representing types of Arduino devices. We can see that adding, say, an LED device could simply be achieved by specializing the `ArduinoDigitalModule` class. Similarly, new types of boards like Raspberry Pi[1] boards can be added by specializing the abstract class `Board`.

3.2.2. Graphical concrete syntax

We implemented a graphical concrete syntax using Emfatic for the conceptual metamodel. More specifically, we use EuGENia annotations in the Emfatic specification (ref. Section 2.1.3).

```
@gmf
@namespace(uri="http://www.example.org/arduinoConfiguration", prefix="arduinoConfiguration")
package arduinoConfiguration;
@gmf.diagram
class Project {
  @gmf.label(label.pattern="projectName = {0}")
  attr String projectName;
  val Board board;
}
abstract class Board {
}
@gmf.node(figure="svg", svg.uri="platform:/plugin/ca.iro.umontreal.geodes.arduino.editor.
    configuration/svg/arduinoBoard.svg",
size="150,150", label.placement="none", margin="0")
class ArduinoBoard extends Board {
  @gmf.link(label="analogPin")
  val AnalogPin[*] analogPins;
  @gmf.link(label="digitalPin")
```

[1]Raspberry Pi: `https://www.raspberrypi.org/`

```
18   val DigitalPin[*] digitalPins;
19   @gmf.link(label="communicationPin")
20   val CommunicationPin[*] communicationPins;
21 }
```

Listing 3.1. Excerpt of the annotated conceptual metamodel of ArduinoDSL

Listing 3.1 shows the annotated version of the classes `Project`, `Board` and `ArduinoBoard` of the metamodel in Figure 3.1. `Project` is annotated as the root of the metamodel by using the *gmf.diagram* annotation. As we wanted to add the display format (ref. Section 2.1.1.2) of each attribute in the metamodel, we used the *gmf.label* annotation for each of them and defined a pattern as shown on line 8. As defined in Section 3.2.1, the pins are contained in the board. However, we did not want to add them in the diagram as compartments in the ArduinoBoard object but linked to it. Hence why we used the *gmf.link* annotation. We used SVG images to represent the devices since they can be scaled without any quality loss. The complete annotated metamodel is shown in Appendix A.

Fig. 3.2. A sample board model in ArduinoDSL showing the configuration of a Grove with an RFID sensor and an LED

Figure 3.2 shows the model of a board where an RFID is connected to digital pins 8 and 9 of a Grove shield. Also, an LED is connected to the digital pin 5 of the Grove shield.

Since the concrete syntax definition maps each metamodel element to a representation, an extensible design of the conceptual metamodel entails an extensible design of the conceptual concrete syntax.

3.3. Behavioral aspect

In ArduinoDSL, the behavioral aspect contains statements that can be expressed in the `setup` and `loop` sections, such as variable assignments, function invocations, loops, and conditionals. It defines the language for the sketch part.

3.3.1. Metamodel

The current implementation of ArduinoDSL does not support all the functions provided by the Arduino language because the goal of this thesis is to showcase evolution rather than re-implementing the whole Arduino language. Hence, the functions to manipulate numbers, bits, or characters are not supported. The behavioral metamodel also defines basic control structures: `if`, `else`, `for`, `while` and `do...while`. Additionally, it supports user-defined function declarations and invocations. The `Serial` library has also been included in the behavioral metamodel since it's a common library (ref. Section 2.4.3). More complex statements, such as Boolean conditions and basic operations supported by the Arduino language, can be defined in unparsed strings.

In ArduinoDSL, we implemented the behavioral metamodel in the form of a grammar using Xtext [**11**]. The metamodel is implicitly defined in the grammar. The grammar is available in Appendix B

Similarly to the board part, the root of the metamodel is `Project`. The `Setup` production rule is used to set the statements that will be in the `setup` function, while the `InfiniteLoop` is used for the `loop` function. Since the board model should have already been defined, the `Setup` production rule may not be used in a sketch model. As the only difference between the `setup` function and the `loop` function is that the first runs only once while the latter runs infinitely, we created two different rules to emphasize on the `setup` function. The functions from the Arduino Reference are represented as production rules. Additionally, we added six other production rules to implement other functionalities of the Arduino language. As mentioned, complex statements are not supported. Hence, we added the production rule `CustomCode` to allow the user to add C++ code. The statements in a `CustomCode` are not evaluated by the code generator and are added as is in the generated Arduino code. The `Comments` production rule allows the user to add comments in the Arduino code. The `Declaration` production rule is used to declare a new variable and the `AssignmentValues` production rule can be used to assign it a value. The `FunctionDeclaration` and `FunctionCall` production rules are used to declare a new function and call it, respectively. We limited the data types to `String`, `int`, `char`, and `bool`.

Unlike the LED, the RFID reader needs an external library to communicate with the RFID tags. Therefore, we added five functions of the RFID125 library (ref. Section 2.4.3) in the behavioral metamodel: `branch` to specify the pin or the two pins on which the RFID

reader is plugged to, `readCode` to read the UID of an RFID tag, `writeCodes` to register UIDs, `codeIsPresent` to check if an UID was registered, and `clearCodes` to delete the list of the registered UIDs. Their production rules are on lines 14 to 32 of Appendix B.

3.3.2. Textual concrete syntax

Since Xtext was used for the behavioral metamodel, the concrete syntax is textual.

```
1 SETUP {
2   SERIAL begin 9600
3 }
4 FUNCTION blink {
5   DIGITAL write HIGH on led
6   TIME waitMilliseconds 2000
7   DIGITAL write LOW on led
8 }
9 RFID125 readCard on rfidSensor -> String code
10 IF 'code!=""'
11   SERIAL print code
12   CALL blink
13 END:IF
14 RFID125 registerCards "07871946 07388281" on rfidSensor
```

Listing 3.2. A sample sketch model in ArduinoDSL showing an interaction with an RFID and an LED

Listing 3.2 is the sketch part of the board model shown in Figure 3.2. On line 2, we first initialize the serial communication built-in the Grove at a specific baud rate. On lines 4 to 8, we declare a new function `blink` which turns on the LED for 2 seconds. On line 9, when an RFID sensor reads an RFID card, it stores its UID in the variable `code`. On lines 10 to 13, when a card is read, the UID is output through the serial port and the `blink` function is called. On line 14, we register the UID of two RFID cards. The sketch then loops back to line 4.

3.4. IDE generation and Arduino code generation

With the metamodel and concrete syntax defined in the previous sections, we generate the IDE of ArduinoDSL. As ArduinoDSL has two aspects, we have in fact two editors: a graphical one for the board part and a textual one for the sketch part. The pair of models created from these two editors are used to generate the Arduino code.

3.4.1. The graphical editor of ArduinoDSL

The graphical editor is a GMF-based editor generated using EuGENia.

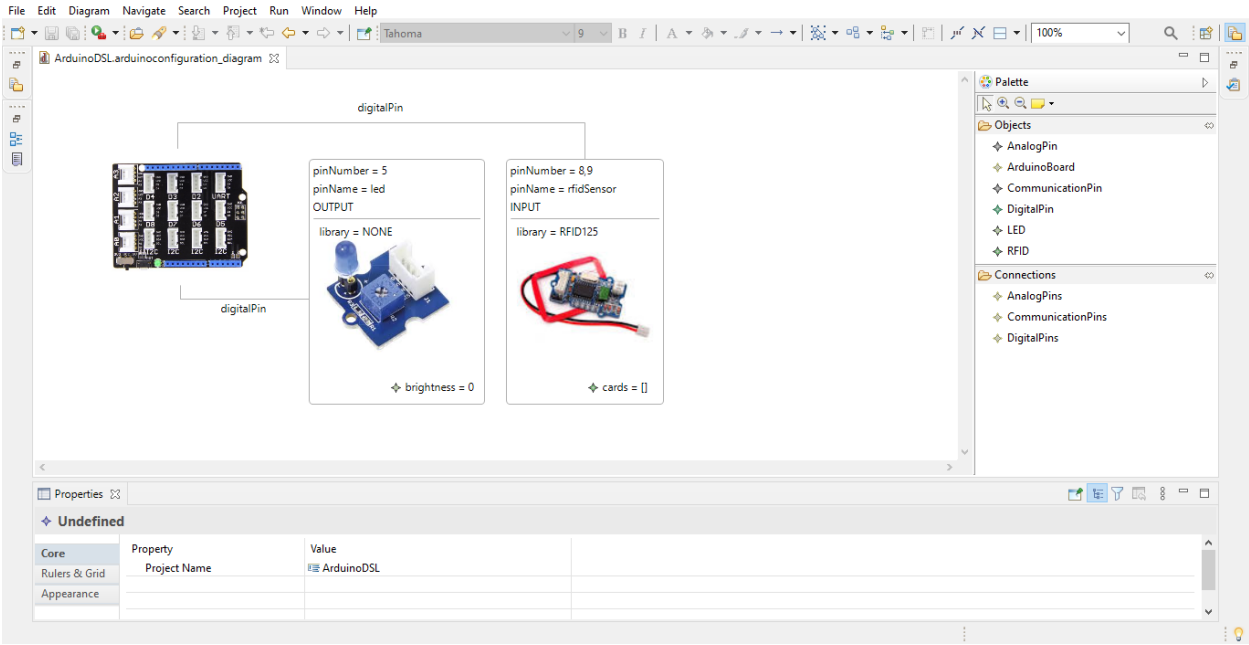

Fig. 3.3. The graphical editor of ArduinoDSL

Figure 3.3 shows an instance of the graphical editor. It is composed of the canvas on which the elements are added and the palette which have the different elements that can be added to the canvas. The palette only contains the elements defined in the conceptual metamodel. Hence, they are only six *objects* and three *connections*. The pin objects (*AnalogPin*, *CommunicationPin*, and *DigitalPin*) are containers. Devices can only be placed inside corresponding containers. For instance, the objects *RFID* and *LED* are devices that should be plugged to a digital pin. Therefore, they are each placed inside a *DigitalPin* container. The only device that will not be placed in a container is the *ArduinoBoard*. It can be placed anywhere in the canvas. To connect the devices to the board, the connections are used. There is a connection for each type of pin. As shown on Figure 3.1, the project should have a name. It can be assigned in the Properties tab at the bottom of the figure.

3.4.2. The textual editor of ArduinoDSL

The textual editor is generated using Xtext. Figure 3.4 shows an instance of the textual editor of ArduinoDSL. The command *Ctrl+Space* allows to have a list of suggestions. The keywords are highlighted in purple. To reduce errors, we only allow the use of the names of the devices already defined in the board model (ref. Figure 3.3). As the board model was not developed using Xtext, cross-referencing is not possible. Therefore, we created a Python script which will add the names of the devices in the Xtext grammar. That is why the devices names are highlighted, as seen on lines 5, 7, 9, and 14 in Figure 3.4.

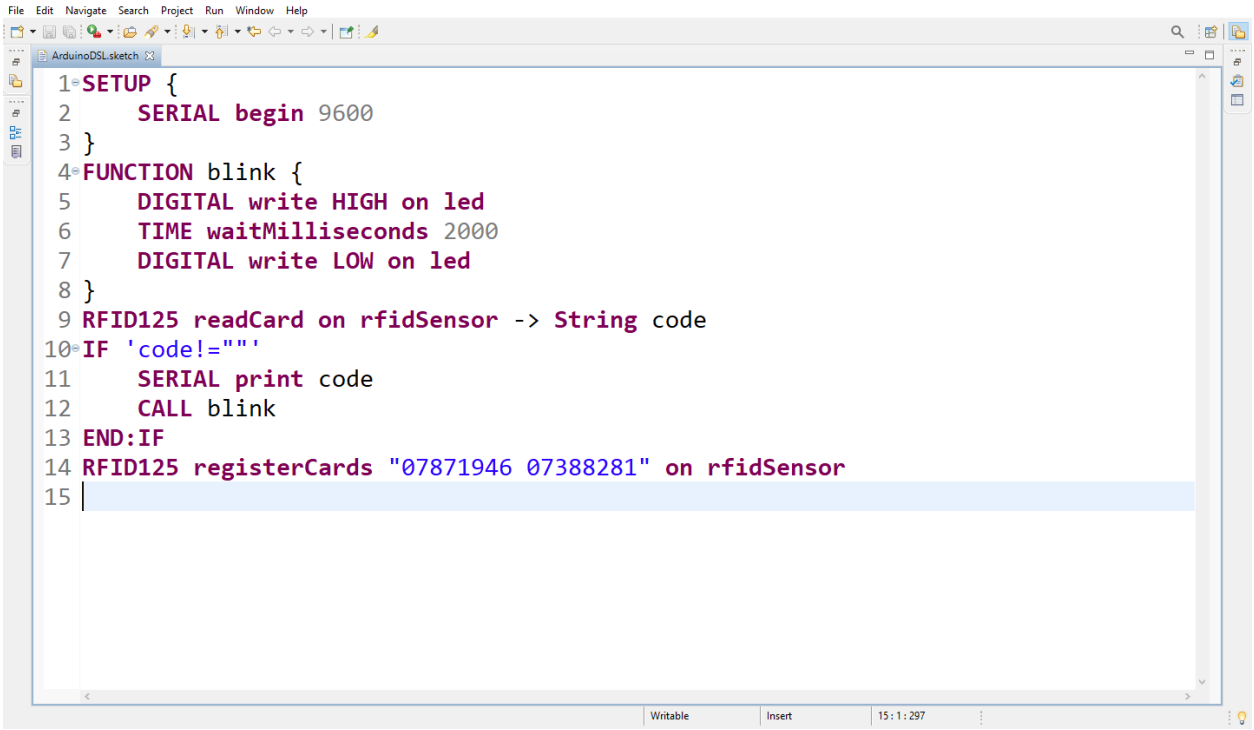

Fig. 3.4. The textual editor of ArduinoDSL

3.4.3. Arduino code generation

We implemented a code generator in EGL for each aspect of the metamodel and used the EGL Coordination Language (EGX) [**73**] to automate the execution.

To map the sketch model to the board model, the sketch model should have the same name as the project name in the board model. The board model code generator is shown in Appendix C.

Since a part of the `setup` function can be defined in the sketch model, the function is not closed in the board model code generator.

Hence, the first lines (lines 1 to 7) from the sketch model code generator in Listing D are to complete the `setup` function. As the `setup` function in the sketch model has the same characteristics as the `loop` function, they are handled in similar ways. Using templates, we define the code to be generated depending on the production rules defined in the Xtext grammar in Appendix B.

```
1 [% operation Sketch!ReadCodeRFID recursive_op() {
2  var fonction = '';
3  if(self.variableName.first().isDefined()){
4   if(self.variableType.isDefined()){
5    fonction = fonction + self.variableType + ' ' + self.variableName.first() + ' = ' + self.
        pin.pins.first() + '.readCode();';
6   } else {
```

```
7       fonction = fonction + self.variableName.first() + ' = ' + self.pin.pins.first() + '.
            readCode();';
8     }
9   } else {
10    fonction = fonction + self.pin.pins.first() + '.readCode();';
11  }%]
12  [%=fonction%]
13 [% } %]
```

Listing 3.3. The template in EGL for the ReadCodeRFID production rule which represents the function readCode defined in the RFID125 library

Listing 3.3 is the template that is used to generate the Arduino code when the *ReadCodeRFID* production rule is used. As the function `readCode` returns the UID of the RFID tag that has been read, the user may have have affected the return value to a variable. The variable may have already been declared or not. If is a newly declared variable, that means that its data type (which can only be `String`) has been given. This is the case evaluated on lines 4 to 6. If the variable was already declared, we branch to lines 7 to 9 for the execution. If no variable was affected to the return value, the lines 11 to 13 are executed.

```
1 #include <RFID125.h>
2 #include <SoftwareSerial.h>
3 RFID125 rfidSensor;
4 const int led = 5;
5 void setup() {
6   Serial.begin(9600);
7   rfidSensor.branch(8,9);
8   pinMode(led, OUTPUT);
9 }
10 void blink() {
11   digitalWrite(led, HIGH);
12   delay(2000);
13   digitalWrite(led, LOW);
14 }
15 void loop() {
16   String code = rfidSensor.readCode();
17   if(code != "") {
18     Serial.println(code);
19     blink();
20   }
21   rfidSensor.writeCodes("07871946 07388281");
22 }
```

Listing 3.4. Arduino code generated from the models in Figure 3.3 and Figure 3.4

The Arduino code generated from the board model in Figure 3.3 and the sketch model in Figure 3.4 is listed in Listing 3.4. This code can be directly uploaded on the Arduino board.

3.5. Anticipated evolution issues

New Arduino devices are added and updated regularly[2]. Therefore, the language engineer will have to add the new devices and libraries in the conceptual metamodel and behavioral metamodel, respectively. However, updating all the artifacts of the DSL for every existing device and library is not scalable. Since external libraries are in constant evolution, we must support evolving APIs and devices. Thus, this also impacts other components of the DSL, such as the code generator. Function invocation and even logic of the generated statements may need to change. Furthermore, the language engineer may want to simplify some function calls for better abstraction by, for example, renaming a function like `println` to `print` or `readCode` to `readCard`.

In the current state of practice in MDE [**59**], the language engineer has to modify the metamodel and concrete syntax and regenerate the DSL every time there is a change in a library or a device. It is not conceivable to manually change the DSL at this rate for every new device or function available. Additionally, representing library functions in a metamodel is like hardwiring function calls in the syntax of a language, which is overly static and inflexible. Therefore, we need to bind the libraries dynamically to the metamodel and automate the evolution process.

[2]List of Arduino devices: `https://arduinomodules.info/`

Chapter 4

Architecture of the evolutionary process

The evolution of an external library is often synonymous with the evolution of the DSLs that depend on it. Rebuilding the DSLs and regenerating their artifacts manually may lead to errors and inconsistencies.

Therefore, automating the integration of the new functionalities and concepts is a possible solution to reduce the manual changes to the current structure of the DSL and its associated artifacts. To this end, we propose a process that minimizes the language engineer's manual effort and is completely transparent to the DSL user.

In our approach, depicted in Figure 4.1, the language engineer defines the specification of the external library in the form of a model and/or encapsulates the functionalities of the external library. This *extension library* represents the concepts to extend the metamodel of the DSL, its associated concrete syntax, as well as any functions the language engineer wishes to expose in the DSL. From the extension library, we generate the different components required for the DSL and associated artifacts to automatically use the library.

In this chapter, we present an overview of the different concepts (represented as boxes in Figure 4.1) used in the architecture of our approach. We will detail each step (represented as arrows in Figure 4.1) in Chapter 5.

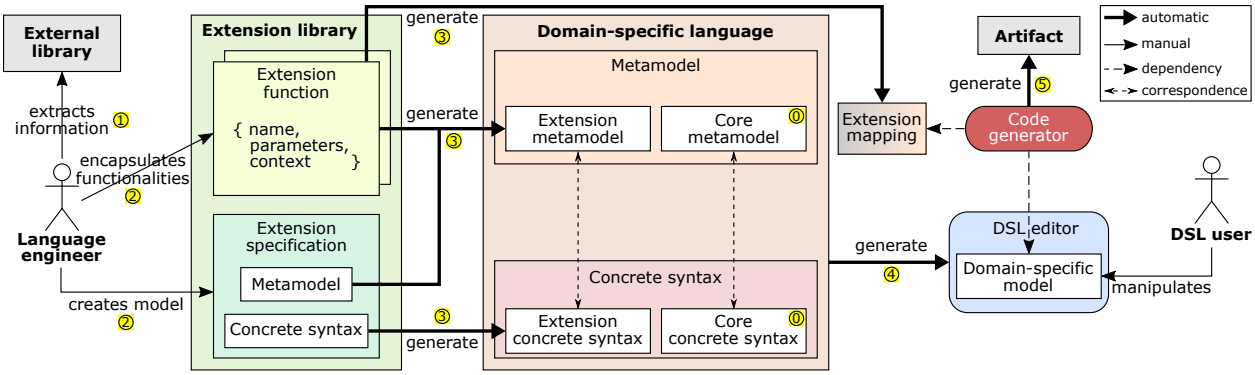

Fig. 4.1. The overall process to evolve a DSL from external libraries

4.1. Specifying the extension library

As shown in Figure 4.1, the language engineer has first to analyze the external library to extract the information relevant to the DSL. She then defines the **extension library** that serves as specification for integrating the external library into the DSL and associated artifacts. The extension library comprises two parts: the **extension specification** and the **extension functions**.

4.1.1. The extension specification

The *extension specification* represents the new concepts that will be part of the DSL. The language engineer specifies the extension at the abstract and concrete syntax levels. Aside from the concepts, the metamodel of the extension specification may contain the set of attributes of the extension concept and the set of relations between the extension concept and existing concepts in the core language. The extension concept can be a sub-type, the target of an association to a core concept or be contained in a core concept. The extension concrete syntax defines the textual and/or the graphical representation of the extension metamodel. It is specific to the technology adopted.

To illustrate, let's take the example defined in Figure 3.2 of a board model in ArduinoDSL. As mentioned in Section 3.2.1, the devices are considered as the main concepts in ArduinoDSL. Therefore, the language engineer can define an extension specification for the RFID and the LED. The creation process of the extension specification models will be detailed in Section 5.1.1.

4.1.2. The extension functions

The second component of the extension library is the set of **extension functions** that encapsulate the functionalities of the concept. However, these functionalities are often implicitly defined in the API of the external library. For example, the RFID can read and write the codes of the cards. Therefore, the language engineer must specify which functions will be available in the DSL so they can be properly invoked via the code generator. Extension functions encapsulate all the necessary information to invoke the corresponding function from the API. Therefore, the language engineer has to extract its name, arguments and context to create the extension function. It is intuitive to know what the name and parameters of a function are. However, it may be a little more complex to grasp what the context of a function entails. The *context* element is a set of constraints. These constraints are a necessary condition to be allowed to use the function. The context information varies from one domain to another and even from one external library to another. Therefore, we let the language engineer define the context of extension functions. For example, in Arduino, when an external library is related to a specific type of device

(e.g., the RFID125 library is specific to RFID sensors), the pin is declared in the constructor of the library. The functions of that external library are then associated with an instance of a class (e.g., *rfidSensor.readCode()*). Hence, in ArduinoDSL, we use the context to hint the code generator on how the external function should be invoked. Context information is useful in other domains as well. For example, suppose a DSL is used to define the communication with micro-services. In this case, each REST API function may require a context specifying the address of the server where the micro-service is deployed. The extraction and encapsulation of the extension functions will be detailed in Section 5.1.2.

The extension functions and extension specification are used to extend the DSL and its associated artifacts automatically.

4.2. Extending the domain-specific language syntax

To extend the DSL, we focus on the two main components of its definition: the metamodel and the concrete syntax.

4.2.1. The extension metamodel

Although DSLs are prone to evolve more frequently than general-purpose languages [**90**], we consider the part of the language that is independent from external libraries to be more stable, with respect to evolution, than the part related to external libraries. For example, the sketch part of ArduinoDSL depends on the Arduino programming language based on C++. On average, there is a new version of C++ every four years, and most changes do not concern the subset used by Arduino. On the contrary, new Arduino libraries appear much more frequently. Therefore, we separate the definition of the DSL into the **core metamodel** and the **extension metamodel**. The former gathers all stable concepts, while the latter gathers all the new concepts that depend on external libraries. For example, in ArduinoDSL, the RFID concept will be part of the extension metamodel, while the concept of TIME (`TIME waitMilliseconds` in Listing 3.2) is part of the core metamodel. The extension metamodel can be generated automatically from the extension specification and extension functions described in Section 4.1. The generation process of the extension metamodel is detailed in Section 5.2.1 and Section 5.2.2.

Ultimately, the DSL must have one metamodel, which is the merge of the core and extension metamodels. Here, we assume the metamodels are described in a meta-language capable of composing metamodels [**6**], such as UML package merge [**25**] or importing a grammar in Xtext. Merging the two metamodels will not yield to conflicts since they do not share overlapping concepts: the extension metamodel simply adds new concepts to the core

metamodel. This requires that the core metamodel is structured in a way to be extended following SOLID design principles. The merged metamodel should not make breaking changes to the core metamodel. It is important to note that our process does not modify the core metamodel that was manually constructed by the language engineer. The merge operation is detailed in Section 5.2.4.

4.2.2. The extension concrete syntax

Similarly to the metamodel, we separate the core from the extension concrete syntax. Since a concrete syntax is the representation of a metamodel, the non-overlapping composition property also holds. The extension concrete syntax can be generated from the extension specification. The generation process is detailed in Section 5.2.3.

The new metamodel and concrete syntax of the evolved DSL can then be used to synthesize a new version of the DSL editor. In this process, the backward compatibility of all models developed by a DSL user is guaranteed by construction. The only exception is when there is a change in the API of the external library that is used by a model. Errors will only be detected when the genereated code from the model will be used.

4.3. Extending the domain-specific semantics

Section 4.2 gave an overview of how the syntax of the DSL is extended. However, the external library also enriches the semantics of the DSL. The semantics of the DSL can be defined in several ways [**35**]. In this paper, we focus on translational semantics defined by means of a code generator.

4.3.1. The code generator

The code generator embeds the semantics of each concept of the DSL and their combinations. For example, in ArduinoDSL, it translates a model to C++ code executable in Arduino. The code generator should not only define the semantics of concepts found in the core metamodel, but also those in the extension metamodel. Therefore, it must be aware of the extension functions.

Our process does not modify the code generator that was manually constructed by the language engineer. However, it must be extensible to be able to handle the extension functions. The generation process of the evolved code generator is detailed in Section 5.3.2.

4.3.2. The extension mapping

To keep the code generator independent from external libraries, we generate an **extension mapping** model from the extension functions. This model maps each extension

function to the corresponding API function to be invoked with all the necessary parameters. Essentially, the extension mapping encodes the semantic mapping between the extension metamodel and the external library. Its construction will be described in Section 5.3.1.

Chapter 5

Evolving DSLs with extension libraries

In Chapter 4, we presented the architecture of our evolutionary process. It is composed of five steps (ref. Figure 4.1). First, the language engineer has to extract the relevant information from the external libraries and create an extension library based on the information collected. From the extension library, the extension metamodel and extension concrete syntax are generated. An extension mapping is also generated that will be used by the code generator to evolve and integrate the extension functions. Each of these steps will be detailed in this chapter using ArduinoDSL as a running example.

5.1. Extracting the information

As mentioned before, the first step to evolve the DSL is to extract the information from the external libraries and create an extension library. The extension library serves as the specification to generate the extension metamodel and concrete syntax to be integrated into the DSL. It is composed of the extension specification, which focuses on the conceptual aspect of the language, and the extension functions, primarily focusing on the behavioral aspect of the language. For each external library, the language engineer shall define an extension specification and encapsulate the extension functions.

5.1.1. Creating the extension specification model

The extension specification defines the domain concept that the external library contributes to in the DSL. It also specifies how the new concept integrates with the core metamodel. Furthermore, it defines the concrete syntax of the concept.

```
1 Concept:
2   'CONCEPT' conceptName = ID '{'
3     ('ATTRIBUTES {' ((attributes += Attribute)(',')?)+ '}')?
4     'RELATION' relation = (SubType | Reference | Containment)
5     ('REPRESENTATION' representation = (Textual | Graphical))?;
```

```
6   '}'(',')?;
7 Attribute: attrName = STRING ':' attrType = AttributeType;
8 SubType: 'EXTENDS' super = STRING;
9 Reference: 'REFERS_FROM' ref_from = STRING;
10 Containment: 'CONTAINED_IN' container = STRING;
```

Listing 5.1. Excerpt of the grammar for the extension specification

We have created a small DSL using Xtext, depicted in Listing 5.1, to help the language engineer in this task. The `conceptName` is a unique identifier that represents the concept C_e of the external library. A concept can hold an arbitrary number of attributes. There are three (3) ways to relate the concept C_e to a concept C_c of the core metamodel. `SubType` allows C_e to extend C_c through sub-typing. `Reference` allows C_e to be the target of an association from C_c. `Containment` allows C_e to be composed within C_c. These three types of relations ensure that integrating the extension metamodel with the core metamodel follows SOLID principles. Note that C_e must be related with C_c using exactly one relation type. Finally, the `representation` specifies the concrete syntax definition for C_e. We currently support a textual concrete syntax through an Xtext grammar or a graphical concrete syntax by means of EuGENia annotations.

```
1 CONCEPT "RFID" {
2   ATTRIBUTES { "cards" : String[*] }
3   RELATION EXTENDS "ArduinoDigitalModule"
4   REPRESENTATION SVG(SVG_URI : "platform:/plugin/svg/rfid.svg")
5 }
6 CONCEPT "LED" {
7   ATTRIBUTES { "brightness" : int }
8   RELATION EXTENDS "ArduinoDigitalModule"
9   REPRESENTATION SVG(SVG_URI : "platform:/plugin/svg/led.svg")
10 }
```

Listing 5.2. Extension specification of RFID125

In ArduinoDSL, the extension specification extends the core metamodel and concrete syntax of the board. Listing 5.2 shows the extension specification for the RFID and LED concepts in ArduinoDSL. To add the new concept RFID, the language engineer defines its name on line 1. She then defines on line 2 the properties associated to the concept. In the case of an RFID, a list of cards can be registered. As an RFID reader is an input device that can be connected via two digital pins to the board, it is a sub-type of the `ArduinoDigitalModule` class, as shown on line 3. Finally, on line 4, the language engineer defines the concrete syntax of the RFID concept as the path to an SVG image. Similarly, she defines the concept of LED, adds the property *brightess* and a concrete syntax (lines 6 to 10).

5.1.2. Extracting and encapsulating the extension functions

The external library may also offer an API as a set of external functions. The language engineer may wish to extend the DSL with an abstraction of the external functions. Thus, she must capture her selection of the external functions in the API that will be integrated into the behavioral aspect of the DSL. Since other systems very likely use the external library than the DSL at hand, the information extraction method from the API to the extension functions should be non-intrusive. One way to extract the information from the API without changing its behavior would be to add annotations or comments ignored by the compiler. For example, for open-source libraries defined in a progamming language like C++, we can rely on C++ attributes [**76**] to annotate each external library function and capture the information needed. Since C++17, all attributes unknown to an implementation are ignored without compilation or run-time error. Alternatively, if the API is not directly modifiable (e.g., the language engineer does not have write access to it, or it is stored on a third-party server), a script can read the API and output the required information for each function. Tools such as cURL [**3**], HTTPie [**4**], and Hurl.it [**5**] allow developers to query APIs.

We define an extension function to be in the form $EF = \langle function, name, parameters, context \rangle$ following the requirements given in Section 4.1.2. $function$ is a pointer to the external function EF represents. The *name* is a label that identifies EF uniquely within the extension library. If the language engineer desires to offer some or all of the parameters of $function$, she can define *parameters* as a set of couples consisting of the name and type of each parameter. If there is a certain *context* in which $function$ has to be ultimately invoked in the final generated code, it must be specified.

As explained in Section 4.1.2, the definition of a context is specific to the DSL. For ArduinoDSL, we use the context to hint the code generator on how the external function should be invoked. Some functions can be invoked directly by name, like `digitalWrite` on line 14 of Listing 3.4. Others require to be invoked in the scope of an object, like `rfidSensor.branch(8,9)` on line 9 of Listing 3.4. In the former case, we consider the function to be explicitly invokable. In contrast, in the latter case, it is implicitly invokable through an object. Additionally, some functions do not depend on any device (e.g., `delay(2000)` on line 15 of Listing 3.4). Therefore, for ArduinoDSL, there are three possibilities for the context: either there is no context, or it is specified with a Boolean value where true means the function is implicit and false means it is explicit.

```
1 class RFID125 {
2   public:
3     [[name('plug'), param('pin1','int'), param('pin2','int'), implicit(true)]]
4     void branch(int8_t pin1, int8_t pin2=-1);
5     [[name('readCard'), implicit(true)]]
6     String readCode();
```

```
7     void writeCodes(String code);
8     bool codeIsPresent(String code, bool lowLevel=false);
9 };
```

Listing 5.3. Excerpt of the annotated RFID125 library

Listing 5.3 is the annotated version of the RFID125 library in Listing 2.7. All functions of the library require a pin to be called implicitly. Line 3 shows the `branch` function that requires pins as parameters. Line 5 is an example of a function with no parameters. On line 9, the *codeIsPresent* function is not annotated because the language engineer did not wish to offer this function to the DSL.

In Arduino, when an external library is related to a specific type of device (e.g., the RFID125 library is specific to RFID sensors), the pin is declared in the constructor of the library. Therefore, the context can only be implicit in those cases. However, the functions defined in the core behavioral metamodel are all independent from the devices. Hence, they either have an explicit context or no context at all. Thus, in ArduinoDSL, a function does not have a context if it is not related to a device.

```
1 [[name('begin'), param('speed','int')]]
2 void begin(long speed);
3 [[name('print'), param('message','String')]]
4 String print(String message);
5 [[name('waitMilliseconds'), param('time','int')]]
6 void delay(unsigned long ms);
7 [[name('write'), param('pin','int'), param('value','int'), implicit(false)]]
8 void digitalWrite(int pin, int value);
```

Listing 5.4. Some annotated functions from the default Arduino Reference library

Recall from Section 3.3 that the core metamodel already includes common libraries. Hence, there wasn't a need to annotate them. However, to show how they can be annotated. Listing 5.4 lists the annotated functions from the core metamodel used in Listing 3.2. The function `begin` on line 2 is an example of a function without a context. It is a function from the `Serial` library, which is independent of all devices. The behavior of the function `digitalWrite` on line 8 depends on the pin but it is not tied to one type of device. Thus, it has an explicit context, and the pin is declared as a parameter.

5.2. Extending the syntax

In Section 5.1.1, we explained what the language engineer must perform to create the extension library. The extension library is the input to the automated process of evolving the DSL to include external libraries. This section explains how the extension metamodel

and extension concrete syntax can be created automatically and integrated with the core metamodel and core concrete syntax of the DSL.

5.2.1. Extending the conceptual metamodel

The extension metamodel is generated from the extension specification and the extension functions. To easily integrate with the core metamodel, we implement the conceptual aspect of the extension metamodel in Emfatic. We implemented code generation templates in EGL to generate the Emfatic code from an extension specification model conforming to Listing 5.1. The code generator creates a concrete class for the concept. If the concept is a sub-type of a core concept, the code generator adds an inheritance relation to the specified concept using the `extends` keyword in Emfatic. If the concept is the target of an association with a core concept, it adds a relation using the `ref` keyword from the core concept to the concept. In the case of a containment relation, it uses the `val` keyword. The attributes of the concept are added to the class using the `attr` keyword.

```
1 @gmf.node(figure="svg", svg.uri="platform:/plugin/svg/rfid.svg", size="150,150", label.placement
      ="none", margin="0")
2 class RFID125 extends ArduinoDigitalModule {
3   @gmf.label(label.pattern="cards = {0}")
4   attr String[*] cards;
5 }
```

Listing 5.5. Generated extension metamodel and concrete syntax for the concept RFID125

Listing 5.5 shows the generated Emfatic code for the extension specification defined in Listing 5.2. On line 2, a class represents the RFID125 concept and extends the `ArduinoDigitalModule` class. Line 4 shows the only attribute defined for the concept.

5.2.2. Extending the behavioral metamodel

We generate the behavioral aspect of the extension metamodel from the extension functions. To easily integrate with the core metamodel, we implement the behavioral aspect of the extension metamodel in Xtext because it is most often used for textual DSLs emulating the invokation of functions of the external library. The order in which the files and the extension functions are processed is unimportant. Since the specification of the extension functions depends on the DSL, a custom code generator to produce the Xtext grammar is needed. For ArduinoDSL, we developed a custom code generator in Python that reads annotated C++ header files representing the extension functions of the library and outputs an Xtext grammar. Recall that for ArduinoDSL, the behavioral aspect is defined in the sketch. The code generator fetches all header files in the Arduino external library directory.

It captures all the C++ attributes it finds, assuming they conform to the notation presented in Listings 5.3 and 5.4.

```
1 Function: RFID125;
2 RFID125: BranchRFID125 | ReadCodeRFID125 | WriteCodesRFID125;
3 BranchRFID125: 'RFID125' 'plug' pin1=INT 'and' pin2=INT 'on' pin=Pins;
4 WriteCodesRFID125: 'RFID125' 'writeCodes' codes=STRING 'on' pin=Pins;
5 ReadCodeRFID125: 'RFID125' 'readCard' 'on' pin=Pins;
```

Listing 5.6. Generated extension metamodel and concrete syntax from the extension function for the RFID125 library

Listing 5.6 shows the generated Xtext grammar for the extension functions defined in Listing 5.3. The code generator creates production rules for each external library. In Section 4.3.1, we mentioned that the language engineer should prepare hooks in the core metamodel to allow the integration of extensions. In our implementation, it suffices to have a class named `Function` which will regroup all extension functions. This translates into a production rule `Function` in Xtext, which lists all the extension functions that are annotated in the external library. To avoid ambiguities in the grammar, we ensure each function is unique by transforming each function name as follows: we capitalize the first letter of the function name and suffixed it with the name of the external library. In our implementation, parameters are simply concatenated with the `and` keyword. Also, we generate the *context* of the extension function only if it is specified. When it is the case, we add the context to the production rule of the extension function using the `on` keyword.

5.2.3. Extending the concrete syntax

We generate a generic extension concrete syntax that the language engineer can customize as she sees fit for her DSL. The extension specification specifies the concrete syntax. For the conceptual aspect, the concrete syntax is not mandatory (see line 5 in Listing 5.1). If it is not provided, a default Xtext grammar is generated from the specification in Listing 5.1. Otherwise, if `representation` is set to textual, the language engineer has to provide an Xtext grammar. For a graphical concrete syntax, the code generator adds EuGENia annotations into the Emfatic code like in Listing 5.5. For RFID125, the concrete syntax specified is in Listing 5.2 and Figure 3.2 shows the result on a sample model using it.

As for the behavioral aspect, thanks to Xtext, the metamodel and textual concrete syntax are defined from the grammar. Lines 4 and 11 in Listing 3.2 show the grammar in action on a sample model using the extension functions of RFID125.

5.2.4. Merging the extension and the core

The generated extension metamodel and concrete must be integrated with the core DSL components. In general, the integration can take various forms depending on the formalism for core and extension specifications, and how they should be integrated. As described in [**6**], there are different techniques to compose DSLs. In our implementation, the integration is performed by merging the extension and core metamodels and concrete syntaxes, as outlined in Section 4.2.

As outlined in Section 4.2, concepts of the extension metamodel should not overlap with concepts of the core metamodel. Typically, the external library adds new concepts or functionality to the core language. However, an external library may redefine a function already available in the core language. In this case, we must ensure the user can unambiguously refer to either implementations. It is possible that conflicts arise between concepts or functions of two external libraries. For example, in Arduino, there are multiple libraries available to manipulate an RFID sensor.

In our implementation, we prevent conflicts between concepts by enforcing a unique concept name to each extension specification (Listing 5.1). To merge the extension specification with the core metamodel and other extensions, we append the newly generated classes to the Emfatic model. With EuGENia annotations, the Emfatic model is also contains the merge of the concrete syntax. For the behavioral aspect, we merge the generated Xtext grammar from the annotated external library with the core grammar using the *import* operation. The language engineer must ensure that she annotates each extension function with a unique name within the same library.

With the merge completed, the result is an evolved DSL extended with concepts and functionalities from selected external libraries. The language engineer can then rely on the usual generators of the language workbench, like EMF, to generate an editor for domain users. Users can then create and edit domain-specific models seamlessly: all previous models are compatible with the evolved DSL, as long as no changes to existing external libraries have been introduced.

5.3. Extending the semantics

As outlined in Section 4.3, the extension specification has an impact on the semantics of the DSL. The extension mapping defines the relation between the extension specification and the semantics of the external library. For ArduinoDSL, it is used to evolve the code generator to invoke the appropriate API of the new external library.

5.3.1. Generating the extension mapping

The extension mapping is composed of the concept mapping, the function mapping, and the parameter mapping. The concept mapping maps the concept name to the external library path. The function mapping maps each extension function to the external function name. The parameter mapping maps each parameter name and type of the extension function to the corresponding parameter in the external library. Note that the parameter mappings are ordered following the signature of the external function.

```
1 <concept name="RFID125" library="path/RFID125.h">
2   <function implicit="true" grammarName="BranchRFID125" name="branch">
3     <parameter grammarName="pin1" grammarType="int" name="pin1" type="int8_t" />
4     <parameter grammarName="pin1" grammarType="int" name="pin2" type="int8_t" />
5   </function>
6 </concept>
```

Listing 5.7. Excerpt of the extension mapping for the RFID125 library

Listing 5.7 illustrates the extension mapping generated from the annotated library in Listing 5.3 for the `branch` function. In our implementation, we represent this mapping in an XML file. It can be generated during the generation of the extension metamodel.

5.3.2. Evolving the code generator

Evolving the semantics of the DSL is specific to each DSL. We implemented the code generator in EGL. We chose this tool because it is a template-based code generator that allows for polymorphism [**56**]. This is a useful feature so that the language engineer does not have to rewrite the templates for each extension. In Section 5.2.2, we showed that extensions are integrated by sub-typing the `Function` abstract class which is the hook to connect to the core metamodel as mentioned in Section 4.3.1. Thanks to polymorphic templates, any template applicable to `Function` is also applicable to its sub-types. The code generator relies on the extension mapping to determine which external function to print when it encounters an extension function. Therefore, for each extension function encountered in a given model, the code generator searches for the concept and function name in the extension mapping.

For ArduinoDSL, the semantics of a model is defined by the Arduino code it corresponds to. Recall from Section 3.4.3 that ArduinoDSL ships with a C++ code generator. This code generator is implemented for the behavioral aspect of the core metamodel. However, now that new concepts and functionalities are added to the DSL, it must also generate the appropriate code for the extension metamodel.

In Listing 5.7, the `grammarName` corresponds to the production rule name parsed by Xtext. Then, the code generator uses the `name` that corresponds to the external function. If the function requires parameters, they are processed in the same order as they appear in the

extension mapping. When the context is implicit (on line 2), the code generator uses the dot notation when printing the external function invocation. For instance, on line 4 of Listing 3.2, the `readCard` function is used from RFID125. Since this function has an implicit context, the corresponding invocation is `rfidSensor.readCode()`, as shown on line 13 of Listing 3.4.

The code generator does not only depend on the behavioral extensions. For example, the model in Figure 3.2 shows that the RFID sensor is connected to pins 8 and 9. The code generator outputs line 9 in Listing 3.4, which shows the call to the `branch` function in the `setup` block.

Chapter 6

Validation

In this chapter, we show the feasibility and usefulness of our approach with a case study on evolving interactive modeling editors that rely on ArduinoDSL. We also discuss the limitations of our approach.

6.1. Case study

We wanted to verify that our evolution approach is applicable in a real setting. Therefore, we validate the feasibility, applicability and usefulness of our approach with a case study on incrementally developing a domain-specific modeling editor.

6.1.1. Synthesis of interactive modeling editors

In previous work [**79**], Sousa et al. proposed a methodology to define modeling editors with customizable user interactions. One of the possible customizations concerns the choice of I/O devices used to interact with the editor. Traditionally, users interact with modeling editors via keystrokes or mouse movements and clicks. However, this may not be optimal for some domains where a dedicated interaction device helps improve the domain user's productivity. The technique in [**79**] proposes multiple viewpoints to customize the editor: the interface model, the interaction model, and the event mapping model. The interface model defines the layout as well as interaction streams and devices that can be used. The interaction model defines the behavior of each interaction in terms of actions on the editor, the model, and the devices. The event mapping model maps each interaction defined in the interaction model to a specific device operation. The three models are then used to generate a web-based editor. For now, only the Google Chrome web browser is supported.

6.1.2. Setup

For our case study, we present the development of a simple editor to configure Pac-Man games where pac-man navigates through grid nodes searching for food to eat, while

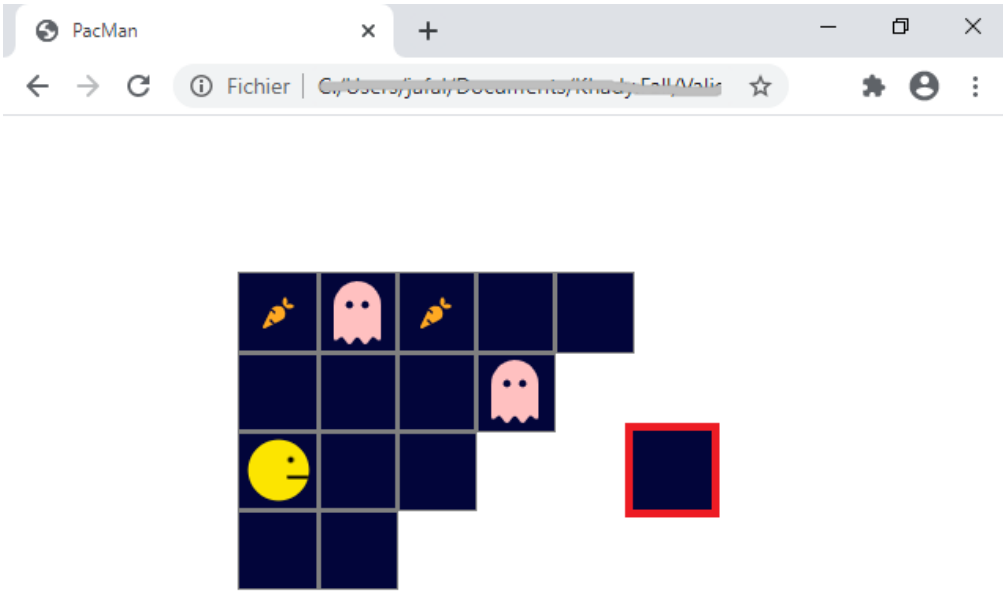

Fig. 6.1. The Pac-man modeling editor in action

ghosts try to kill him [**69**]. Figure 6.1 shows a snapshot of the modeling editor in action. We developed a simple DSL for Pac-Man game configurations, from which we generate the web-based editor. The editor consists of a canvas where language elements can be created, removed, and moved around. However, instead of creating a Pac-Man model with a mouse and keyboard, we built a custom Arduino device for users to interact more naturally with the editor. The editor runs on a web browser communicating with a Node.js server. The Arduino device communicates with the server via a serial port following a simple protocol relying on the Socket.IO library[1]. The device is configured with an Arduino Leonardo board mounted with a Grove shield to simplify the connection with other devices.

Figure 6.2 shows the final Arduino configuration we have built after the last increment. We used ArduinoDSL to develop the reactive behavior of the Arduino (see Appendix F), from which we generated the code to be deployed on the device.

To simplify the case study description, we assume there are three people involved in the project. A gaming expert, Charlie, is the target user of the Pac-Man modeling editor who interacts with the Arduino device to manipulate Pac-Man game models. The language engineer, Bob, has defined the Pac-Man DSL and has developed the ArduinoDSL model that is used by the generated Pac-Man modeling editor. However, Bob is a user of ArduinoDSL and requires using devices and libraries not included in the default Arduino library. Thus,

[1]Socket.IO: `https://socket.io/docs/`

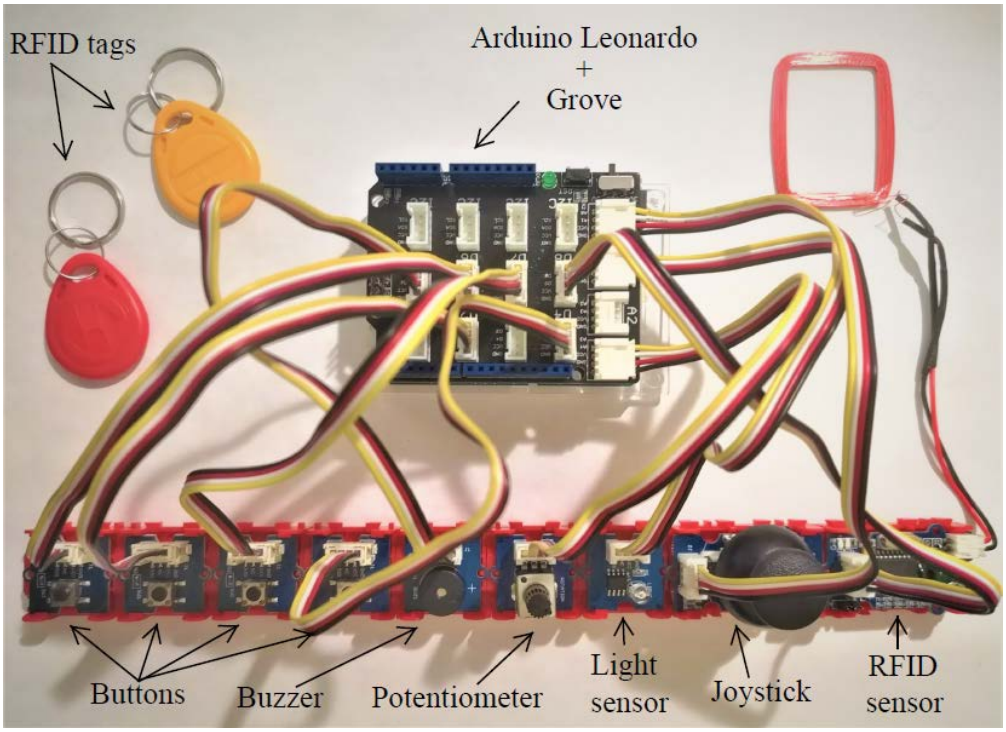

Fig. 6.2. The Arduino configuration for the Pac-man modeling editor

another language engineer, Alice, has developed ArduinoDSL and needs to evolve it according to the devices and external libraries Bob needs. In this case study, we focus on the development of the ArduinoDSL model. Therefore, according to the terminology presented in Figure 4.1, Alice is the language engineer, and Bob is the DSL user.

6.1.3. Incremental evolution

Knowing the devices available to him, Bob developed the model incrementally. Each increment introduces a new device to enhance the feature of the user interaction with the editor. Therefore, at each increment, the ArduinoDSL language needed to evolve, integrating a new concept or external library that was not already available in the language. In the following, we present how they are integrated using our approach.

6.1.3.1. **Creating and moving Pac-Man elements**. In the first increment, Bob wanted to provide the basic ability to create a Pac-Man game model. The Pac-Man DSL consists of four element types: pacman, ghost, food, and grid nodes. Therefore, Charlie does not need a full keyboard to create each type of element. Instead, Bob built an Arduino board with only four buttons to select which element type to create. Since the button device is not part of ArduinoDSL, he asks Alice to provide this feature. Following the procedure defined in Section 5.1.1, Alice creates an extension specification for the `Button` concept (see Listing 6.1).

```
CONCEPT "Button" {
  RELATION EXTENDS "ArduinoDigitalModule"
  REPRESENTATION SVG(SVG_URI : "platform:/plugin/svg/button.svg")
}
```

Listing 6.1. Extension specification for the Button concept

Since the functions needed to use a button are already part of the core ArduinoDSL, she does not need to specify extension functions. Alice then generates a new ArduinoDSL editor extended with buttons. Now Bob can create four buttons in his board model and specify their behavior in the sketch model. When, for example, the button corresponding to ghosts is pressed, the device sends a unique message to the editor via the serial port. In the interaction model Bob created, this instantiates a ghost on the cell where the cursor is at the moment. Figure 6.1 shows a grid node created where the cursor (red box) is located.

In the Pac-Man game, elements can move on locations defined by the grid. Therefore, moving elements freely with a mouse is counter-intuitive. This is why Bob requires the use of a 2-axis joystick to move the cursor left or right and up or down. Joysticks are not part of the Arduino default language. Consequently, Alice creates an extension specification for the `Joystick` concept (see Listing 6.2).

```
CONCEPT "Joystick" {
  RELATION EXTENDS "ArduinoAnalogModule"
  REPRESENTATION SVG(SVG_URI : "platform:/plugin/svg/joystick.svg")
}
```

Listing 6.2. Extension specification for the Button concept

She then selects an external library, like the `Mouse` library[2], to operate the joystick. She annotates the library to define the extension functions required.

```
class Mouse_ {
  public:
    [[name('enable'), implicit(true)]]
    void begin(void);
    [[name('disable'), implicit(true)]]
    void end(void);
    [[name('move'), param('xValue','Character'), param('yValue','Character'), implicit(true)]]
    void move(signed char x, signed char y, signed char wheel = 0);
    [[name('readAxis'), param('axis','int'), implicit(true)]]
    void readAxis(signed char x, signed char y, signed char wheel = 0);
    [[name('pressed'), param('event','Character'), implicit(true)]]
    bool isPressed(uint8_t b = MOUSE_LEFT);
};
```

Listing 6.3. Excerpt of the annotated Mouse library

[2] https://www.arduino.cc/reference/en/language/functions/usb/mouse/

Alice can then generate an evolved ArduinoDSL editor now offering functionalities for buttons and joysticks. Bob added the joystick in its board model and was able to define the behavior in the sketch model.

```
SETUP {
  SERIAL begin 9600
  MOUSE enable
}
DIGITAL read on pacmanButton -> pacmanPressed
DIGITAL read on ghostButton -> ghostPressed
DIGITAL read on foodButton -> foodPressed
DIGITAL read on gridButton -> gridPressed
IF 'pacmanPressed'
  SERIAL print "addPacman"
  TIME waitMilliseconds 500
ELSEIF 'ghostPressed'
  SERIAL print "addGhost"
  TIME waitMilliseconds 500
ELSEIF 'foodPressed'
  SERIAL print "addFood"
  TIME waitMilliseconds 500
ELSEIF 'gridPressed'
  SERIAL print "addGrid"
  TIME waitMilliseconds 500
END:IF
MOUSE readAxis on A1 -> int xValue
MOUSE readAxis on A0 -> int yValue
MOUSE move xValue and yValue and 0
MOUSE pressed "MOUSE_LEFT" -> mousePressed
IF 'mousePressed'
  SERIAL print "selectElement"
  TIME waitMilliseconds 500
  WHILE 'mousePressed'
  END:WHILE
  SERIAL print "moveElement"
  TIME waitMilliseconds 500
END:IF
```

Listing 6.4. The sketch model defining the behavior of the Pac-Man game after the first increment

Listing 6.4 shows the sketch model after the first increment. Charlie can press on the buttons to create language elements. To move a language element, he can use the joystick to move to the element, select it by using the joystick button, then move the element to the desired place. By releasing the joystick button, Charlie has successfully moved the element.

6.1.3.2. **Creating food elements randomly**. Food is one of the most created elements when configuring a Pac-Man game. Thus, using the button and joystick becomes repetitive. Therefore, Bob desires to define a new interaction that creates food randomly on available grid nodes. He wants to use a light sensor that measures the brightness level of the ambient light. Alice extends ArduinoDSL with the `LightSensor` concept by defining it in an extension specification (see Listing 6.5). Since the light sensor only needs to read from an analog pin, no API from an external library is necessary.

```
CONCEPT "LightSensor" {
  ATTRIBUTES {"threshold" : int}
  RELATION EXTENDS "ArduinoAnalogModule"
  REPRESENTATION SVG(SVG_URI : "platform:/plugin/svg/lightsensor.svg")
}
```

Listing 6.5. Extension specification for the LightSensor concept

With the evolved DSL, Bob decides to add a light sensor to the board model and creates food on a random grid node as long as the brightness level is under a certain threshold.

6.1.3.3. **Notifying the user of the creation of food**. Bob wishes to give audible feedback to the user every time food is created in the model. He has a piezo speaker available that produces a tone when an electric current passes through it. Therefore, Alice evolves ArduinoDSL with a `Buzzer` concept for which the functions are already available in the language. Since food can be created manually or automatically, the specification in the sketch model must appear twice. Bob encapsulates these instructions into a function called `scream`, which can be invoked where appropriate to avoid duplication (see Listing 6.6). As opposed to the previous increments, this evolution of ArduinoDSL allows for output devices to receive a trigger from the Pac-Man modeling editor.

```
FUNCTION scream {
  SERIAL available -> int freeSerial
  IF 'freeSerial > 0'
    SERIAL read -> String message
    IF 'message=="scream"'
      TONE play 1000 on buzzerPin
      TIME waitMilliseconds 500
      TONE stop on buzzerPin
    END:IF
  END:IF
}
```

Listing 6.6. The scream function to play a sound whenever a food element is added on the grid

6.1.3.4. **Alternating the concrete syntax**. Another game expert, David, is also using the Pac-Man editor. David prefers to model a game in a three-dimensional setting. Therefore, Bob creates a new concrete syntax for the Pac-Man DSL. In the Arduino model, he adds an RFID sensor and assigns one card tag per concrete syntax. This way, Charlie and David can switch between the two concrete syntaxes by tapping the corresponding card on the sensor. Therefore, Alice extended the ArduinoDSL with the RFID concept offering the functionalities from the `RFID125` library, as illustrated in Section 5.1. This increment shows how the evolved ArduinoDSL can also control the behavior of the editor.

6.1.3.5. **Changing the size of the language elements**. Charlie would like to enlarge or shrink language elements. Bob provides a potentiometer that is equipped with a knob that can be turned, providing a variable resistance. Alice adds the new concept with an extension specification. Although its values can be read through the analog pin, Alice wants to allow the potentiometer to return sizes from 0 to 10, instead of values between 0 to 1023. The `map` function available in the built-in Arduino library can achieve this transformation. However, advanced mathematical operators are not part of the core behavioral metamodel of ArduinoDSL. Therefore, Alice adds the online `Math` external library[3], which re-implements these advanced mathematical functions, and annotates its `map` function. This last increment shows that the DSL can evolve to cover functionalities it did not include even if its semantical domain already did, as shown in Listing 6.7.

```
1 MATH map potentioValue and 0 and 1023 and 1 and 10 -> potentioValue
2 IF 'potentioValue != oldPotentioValue'
3   oldPotentioValue = potentioValue
4   String zoom = "zoom" + potentioValue
5   SERIAL print zoom
6   TIME waitMilliseconds 500
7 END:IF
```

Listing 6.7. The scream function to play a sound whenever a food element is added on the grid

6.1.4. Applicability, feasibility, usefulness

The goal of this case study is to evaluate our approach on three dimensions: its applicability, its feasibility, and its usefulness.

6.1.4.1. **Applicability**. The implementation of this case study shows that the approach is applicable in a real setting. The language engineer can successfully integrate external libraries in the DSL. For ArduinoDSL, these evolutions are those of the conceptual and behavioral metamodels. The approach can manage to add new concepts and behavior,

[3] `https://github.com/arduino/reference-en/tree/master/Language/Functions/Math`

and their modifications. The case study also demonstrates that the language engineer can effectively build a custom I/O device that allows the DSL user to interact with a modeling editor. Nevertheless, our approach is applicable to ArduinoDSL only because its variable part depends exclusively on external libraries. Furthermore, since ArduinoDSL relies on Xtext and Emfatic, it is straightforward to merge the core and the extension DSLs.

6.1.4.2. **Feasibility**. We measure the feasibility of our approach with the easiness of its applicability in a given DSL. As shown in the case study, the needs of the DSL user only require the language engineer to create extension specification models and extract the relevant functions. As defined in Section 5.1.1, the DSL to create extension specification models helps reducing the effort of the language engineer as she does not need to develop a new solution. However, the accessibility of the external libraries impacts the feasibility of the approach.

6.1.4.3. **Usefulness**. The usefulness of our approach is measured by its relevance. As mentioned in Section 3.5, new Arduino devices are added regularly and, to be able to use them, new libraries are developed. At the time of writing, 3 460 external libraries were registered in the Arduino Library Manager[4]. Adding manually each external function in the DSL is laborious, time-consuming, and error-prone. Using our approach, as shown in this case study, the language engineer has to add only the functions in a semi-automatic way. This applies to most DSLs since they are more prone to evolution than general-purpose languages.

6.2. Discussion

We now discuss alternative designs and practical concerns of our approach.

6.2.1. Extracting functions

A primary concern when dealing with external libraries is their accessibility. For Arduino, most libraries are available in open-source. Hence, the language engineer can easily download them on a local server and annotate them, as we have shown in this paper. However, in many practical settings, external libraries remain on a third-party remote server (e.g., Javascript libraries). Therefore, unless the language engineer obtains a local copy of a library, it is not feasible to annotate it through uncompiled attributes or comments. Furthermore, even if a local copy is available, the library may have many other dependencies that would complicate its local build. This is also true for legacy libraries. In this case, the language engineer should seek alternative ways to extract the extension functions from the API. For example,

[4]Arduino Library List: `https://www.arduinolibraries.info/`

as we mentioned in Section 5.1.2, she could develop a script that connects to the remote server, parses the API, and outputs the extension functions in the expected format.

6.2.2. Generating the artifacts

As we have seen with the case study, DSL extensions can be generated at will. In our implementation, there is one extension metamodel (which may be separated for the conceptual and behavioral aspects), one extension concrete syntax (for each aspect), and one extension mapping. This means each time an extension is generated, it is lumped with previous extensions. For example, in the case study, all the new concepts are stored in a single Emfatic model, and all the new extension functions are stored in a single Xtext grammar. Consequently, every time there is a change in an external library or a new one is added, all these artifacts are regenerated. Alternatively, it may be more modular and efficient to generate these artifacts incrementally. In this case, each extension library has its own set of generated artifacts. However, this proliferation of artifacts may require additional effort to manage a log of their location and relations.

6.2.3. Merging the artifacts

One assumption of our approach is that the formalisms to define metamodels and concrete syntax must allow for a merge operation as described in Sections 4.2 and 5.2.4. For example, in our implementation, we use the import operator of Xtext to merge extension and core grammars, and the UML package merge operator to merge extension and core metamodels. The concrete syntax and extension mappings are merged by appending EuGENia annotation and XML elements, respectively. However, the choice of the formalism and its merge operation may impose limitations on how the language engineer can express extension libraries.

6.2.3.1. **Textual grammar**. Using a grammar specification like Xtext imposes some restrictions on the textual concrete syntax. The name of each production rule must be unique, meaning that the name of each external library must be unique. For example, extending ArduinoDSL with two libraries called RFID will create a conflict. This is why, in the running example, we called it RFID125 to distinguish it from others. Another restriction of Xtext is that it is based on an LL(*) parser [**66**], which imposes a specific ambiguity resolution. In particular, this means that extension functions cannot have the same name. Suppose an external function f_1 is overloaded by another function f_2 in the external library. Assume f_2 has one more parameter than f_1. Then, we must ensure that their corresponding extension functions have distinct names. The language engineer should be aware of such restrictions when annotating the functions.

6.2.3.2. **Conceptual metamodel**. In UML, classes with the same name are considered aliases. In Emfatic, classes must have unique names. Since each class represents a concept, a conflict arises if two external libraries share the same concept. For example, the RFID125 and SeeedRFID[5] are two external Arduino libraries that provide an API for an RFID sensor and tags. In our implementation, the extension specification of each external library must have a unique `conceptName` (see Listing 5.1), even if, conceptually, they refer to the same RFID concept. The specification of relations may also be a source of conflict. Since each relation type is transformed into a UML association, we must ensure well-formedness rules are still satisfied. For instance, sub-type and containment relations must be transitive and cannot be circular. In principle, each relation should connect a class from the core metamodel and a class from the extension metamodel. Relations between classes of the core metamodel should be prohibited as they do not express an extension from a new concept. A relation between classes of the extension metamodel means that their corresponding external libraries depend on each other. In the current implementation, we assumed external libraries are independent of each other.

6.2.3.3. **Graphical concrete syntax**. For graphical concrete syntax, one requirement is that each concept from the core and extension metamodel has a distinguishable representation to avoid ambiguities. Moody [**61**] defines an extensive set of principles for designing an adequate graphical concrete syntax, which the language engineer should follow.

[5]SeeedRFID library: `https://github.com/Seeed-Studio/RFID_Library`

Chapter 7

Conclusion

We conclude by summarizing the contributions of this thesis and outlining future work. The work presented in this thesis makes several contributions to the field of automation in MDE.

7.1. Summary

A DSL represents the concepts of a specific domain. As the domain evolves, the DSL has to evolve accordingly. Many approaches were proposed in the literature for the evolution and co-evolution of DSL components (ref. Section 2.2). Since DSLs are restricted languages, they often rely on external dependencies which are not necessarily modeled like external libraries. The evolution of the external libraries may impact the DSL. However, no research has been done in integrating the evolution of external libraries in the DSL. The language engineer could only manually add the new features in the DSL.

In our work, we first developed a DSL for modeling with Arduino, ArduinoDSL (ref. Chapter 3). Analyzing the Arduino language and the modeling tools provided (ref. Section 2.4), we have seen that the language heavily relies on external libraries. By developing ArduinoDSL, we could grasp its evolution limits as seen in Section 3.5.

To tackle those limitations, we formulated, in Chapter 4, an approach to integrate the evolution of external libraries in DSLs. The approach separates the DSL in two aspects: the core and the extension. The core regroups the stable concepts while the extension contains the functionalities from the external libraries. By separating the language, we avoid creating a static language since adding library functions in a metamodel is equivalent to hardwiring function calls in the syntax of the language. We minimize the language engineer's manual effort as she will have to perform two tasks: (i) specify the external library by annotating it and creating a specification model for the concepts, and (ii) develop a code generator to produce an Xtext grammar from annotations. The extended language will be generated from that specification resulting in an evolved DSL.

We described the implementation of our approach in Chapter 5. We created an Xtext DSL to help the language engineer define the concepts of the external libraries. As the code generator to produce the grammar from the annotations depends on the DSL, we provide a code generator that generates the grammar from C++ attributes.

In Chapter 6, we show the feasibility of our approach with a case study on evolving interactive modeling editors that rely on ArduinoDSL. The approach can manage to add new concepts and behavior, and their modifications. Thus, we demonstrated that language engineers can successfully integrate external libraries in the DSL.

7.2. Outlook

Currently in ArduinoDSL, the conditions in a conditional structure are strings since complex statements are not supported. A potential extension is to handle boolean constructs. This also ties to the user-defined functions as the return value of a function can be a boolean or used in a condition. However, in its actual state, ArduinoDSL does not allow return values for the functions. Thus, another potential improvement would be the addition of return values for the functions as well as parameters.

Our approach deals with non-breaking changes in external libraries. A potential extension is to handle breaking changes in external libraries, such as the removal of external functions or libraries used in a model. As we currently assume external libraries are independent of each other, another improvement would be to relax this assumption to apply our approach in more complex domains, such as programming languages. For example, the Java API[1] contains multiple inter-library dependencies with libraries using structures from other libraries. In Section 5.2.2, we mentioned that a custom code generator is needed to produce the Xtext grammar. We developed a code generator that creates the grammar from C++ attributes. A potential extension is to have a family of code generators each aimed at a popular language.

[1]Java API Documentation: `https://docs.oracle.com/en/java/javase/15/docs/api/index.html`

Bibliography

[1] `https://www.eclipse.org/gmf-tooling/`. Last access: 11-10-2020.

[2] `https://wiki.eclipse.org/Graphiti`. Last access: 11-10-2020.

[3] `https://curl.haxx.se/`. Last access: 22-09-2020.

[4] `https://httpie.org/`. Last access: 22-09-2020.

[5] `https://www.hurl.it/`. Last access: 22-09-2020.

[6] Abouzahra, A., Sabraoui, A., and Afdel, K. Model composition in model driven engineering: A systematic literature review. *Information and Software Technology 125* (2020).

[7] Anguel, F., Amirat, A., and Bounour, N. Towards models and metamodels co-evolution approach. In *2013 11th International Symposium on Programming and Systems (ISPS)* (2013), IEEE, pp. 163–167.

[8] Banzi, M., and Shiloh, M. *Getting Started with Arduino: The Open Source Electronics Prototyping Platform*. Maker Media, Inc., 2014.

[9] Batot, E., Kessentini, W., Sahraoui, H., and Famelis, M. Heuristic-based recommendation for metamodel-ocl coevolution. In *2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS)* (2017), IEEE, pp. 210–220.

[10] Bellamy-Royds, A., Bah, T., Lilley, C., Schulze, D., and Willigers, E. Scalable vector graphics (svg) 2. *s Draft. https://svgwg.org/svg2-draft/* (2020).

[11] Bettini, L. *Implementing Domain Specific Languages with Xtext and Xtend - Second Edition*, 2nd ed. Packt Publishing, 2016.

[12] Christensen, E., Curbera, F., Meredith, G., Weerawarana, S., et al. Web services description language (wsdl) 1.1, 2001.

[13] Cicchetti, A., Di Ruscio, D., Eramo, R., and Pierantonio, A. Automating co-evolution in model-driven engineering. In *2008 12th International IEEE Enterprise Distributed Object Computing Conference* (2008), IEEE, pp. 222–231.

[14] Daly, C. Emfatic language reference. *IBM alphaWorks* (2004).

[15] Del Fabro, M. D., and Valduriez, P. Semi-automatic model integration using matching transformations and weaving models. In *Proceedings of the 2007 ACM symposium on Applied computing* (2007), pp. 963–970.

[16] Demuth, A., Lopez-Herrejon, R. E., and Egyed, A. Cross-layer modeler: a tool for flexible multilevel modeling with consistency checking. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering* (2011), pp. 452–455.

[17] Demuth, A., Lopez-Herrejon, R. E., and Egyed, A. Supporting the co-evolution of metamodels and constraints through incremental constraint management. In *International Conference on Model Driven Engineering Languages and Systems* (2013), Springer, pp. 287–303.

[18] Demuth, A., Riedl-Ehrenleitner, M., Lopez-Herrejon, R. E., and Egyed, A. Co-evolution of metamodels and models through consistent change propagation. *Journal of Systems and Software 111* (2016), 281–297.

[19] Di Ruscio, D., Iovino, L., and Pierantonio, A. What is needed for managing co-evolution in mde? In *Proceedings of the 2nd International Workshop on Model Comparison in Practice* (2011), pp. 30–38.

[20] Di Ruscio, D., Iovino, L., and Pierantonio, A. Managing the coupled evolution of metamodels and textual concrete syntax specifications. In *2013 39th Euromicro Conference on Software Engineering and Advanced Applications* (2013), IEEE, pp. 114–121.

[21] Di Ruscio, D., Lämmel, R., and Pierantonio, A. Automated co-evolution of gmf editor models. In *International Conference on Software Language Engineering* (2010), Springer, pp. 143–162.

[22] Dig, D., and Johnson, R. The role of refactorings in api evolution. In *21st IEEE International Conference on Software Maintenance (ICSM'05)* (2005), IEEE, pp. 389–398.

[23] Dig, D., and Johnson, R. How do apis evolve? a story of refactoring. *Journal of software maintenance and evolution: Research and Practice 18*, 2 (2006), 83–107.

[24] Dig, D., Negara, S., Mohindra, V., and Johnson, R. Reba: Refactoring-aware binary adaptation of evolving libraries. In *Proceedings of the 30th international conference on Software engineering* (2008), pp. 441–450.

[25] Dingel, J., Diskin, Z., and Zito, A. Understanding and improving UML package merge. *Software and Systems Modeling 7* (2008), 443–467.

[26] Dresden, T. Software technology group: Emftext, 2009.

[27] Efftinge, S. Xtend. *URL: https://eclipse. org/xtend/index. html. Developed by: openArchitectureWare-Eclipse M2T* (2015).

[28] Efftinge, S., Friese, P., Hase, A., Hübner, D., Kadura, C., Kolb, B., Köhnlein, J., Moroff, D., Thoms, K., Völter, M., et al. Xpand documentation. *Eclipse Foundation, Ottawa, Canada, Tech. Rep* (2004).

[29] Erdweg, S., Rendel, T., Kästner, C., and Ostermann, K. Sugarj: library-based syntactic language extensibility. In *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications* (2011), pp. 391–406.

[30] Fokaefs, M., Mikhaiel, R., Tsantalis, N., Stroulia, E., and Lau, A. An empirical study on web service evolution. In *International Conference on Web Services* (2011), IEEE, pp. 49–56.

[31] Fowler, M. Language workbenches: The killer-app for domain specific languages?, 2005.

[32] Garcés, K., Jouault, F., Cointe, P., and Bézivin, J. Managing model adaptation by precise detection of metamodel changes. In *European Conference on Model Driven Architecture-Foundations and Applications* (2009), Springer, pp. 34–49.

[33] Gronback, R. C. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Pearson Education, 2009.

[34] Harel, D. Statecharts: A visual formalism for complex systems. *Science of computer programming 8*, 3 (1987), 231–274.

[35] Harel, D., and Rumpe, B. Meaningful Modeling: What's the Semantics of "Semantics"? *IEEE Computer 37*, 10 (2004), 64–72.

[36] Harrand, N., Fleurey, F., Morin, B., and Husa, K. E. Thingml: a language and code generation framework for heterogeneous targets. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems* (2016), pp. 125–135.

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

[38] Henkel, J., and Diwan, A. Catchup! capturing and replaying refactorings to support api evolution. In *Proceedings of the 27th international conference on Software engineering* (2005), pp. 274–283.

[39] Herrmannsdoerfer, M., Benz, S., and Juergens, E. Cope-automating coupled evolution of metamodels and models. In *European Conference on Object-Oriented Programming* (2009), Springer, pp. 52–76.

[40] Hieronymus, J. L. Ascii phonetic symbols for the world's languages: Worldbet. *Journal of the International Phonetic Association 23* (1993), 72.

[41] Johnson, S. C., et al. *Yacc: Yet Another Compiler-Compiler*, vol. 32. Bell Laboratories Murray Hill, NJ, 1975.

[42] Jouault, F., Allilaire, F., Bézivin, J., and Kurtev, I. Atl: A model transformation tool. *Science of computer programming 72*, 1-2 (2008), 31–39.

[43] Jouault, F., Bézivin, J., and Kurtev, I. Tcs: a dsl for the specification of textual concrete syntaxes in model engineering. In *Proceedings of the 5th international conference on Generative programming and component engineering* (2006), pp. 249–254.

[44] Judson, S. R., France, R. B., and Carver, D. L. Specifying model transformations at the metamodel level. In *Proceedings of the Workshop in Software Model Engineering (WiSME2003), San Francisco, CA, USA* (2003).

[45] Kadhim, B. M., and Waite, W. M. Maptool — supporting modular syntax development. In *International Conference on Compiler Construction* (1996), Springer, pp. 268–280.

[46] Kalleberg, K. T., and Visser, E. Spoofax: An extensible, interactive development environment for program transformation with stratego/xt. *Technical Report Series TUD-SERG-2007-018* (2007).

[47] Kessentini, W., Sahraoui, H., and Wimmer, M. Automated metamodel/model co-evolution using a multi-objective optimization approach. In *European Conference on Modelling Foundations and Applications* (2016), Springer, pp. 138–155.

[48] Khelladi, D. E., Bendraou, R., Hebig, R., and Gervais, M.-P. A semi-automatic maintenance and co-evolution of ocl constraints with (meta) model evolution. *Journal of Systems and Software 134* (2017), 242–260.

[49] Khelladi, D. E., Hebig, R., Bendraou, R., Robin, J., and Gervais, M.-P. Detecting complex changes during metamodel evolution. In *International Conference on Advanced Information Systems Engineering* (2015), Springer, pp. 263–278.

[50] Knuth, D. E. On the translation of languages from left to right. *Information and control 8*, 6 (1965), 607–639.

[51] Kolovos, D. S., García-Domínguez, A., Rose, L. M., and Paige, R. F. Eugenia: towards disciplined and automated development of gmf-based graphical model editors. *Software and Systems Modeling 16* (2017), 229–255.

[52] Kolovos, D. S., Paige, R. F., and Polack, F. A. Eclipse development tools for epsilon. In *Eclipse Summit Europe, Eclipse Modeling Symposium* (2006), vol. 20062, Citeseer, p. 200.

[53] Kramer, D. Api documentation from source code comments: a case study of javadoc. In *Proceedings of the 17th annual international conference on Computer documentation* (1999), pp. 147–153.

[54] Levendovszky, T., Balasubramanian, D., Narayanan, A., and Karsai, G. A novel approach to semi-automated evolution of dsml model transformation. In *International Conference on Software Language Engineering* (2009), Springer, pp. 23–41.

[55] Li, J., Xiong, Y., Liu, X., and Zhang, L. How does web service api evolution affect clients? In *International Conference on Web Services* (2013), IEEE, pp. 300–307.

[56] Luhunu, L., and Syriani, E. Comparison of the expressiveness and performance of template-based code generation tools. In *Proceedings of the 10th ACM SIGPLAN International Conference on Software Language Engineering* (2017), pp. 206–216.

[57] Martin, R. C. Design principles and design patterns. *Object Mentor 1*, 34 (2000), 597.

[58] McGill, M. J. Uml class diagram syntax: An empirical study of comprehension.

[59] Meyers, B., and Vangheluwe, H. A Framework for Evolution of Modelling Languages. *Science of Computer Programming 76*, 12 (2011), 1223–1246.

[60] Meyers, B., Wimmer, M., Cicchetti, A., and Sprinkle, J. A generic in-place transformation-based approach to structured model co-evolution. *Electronic Communications of the EASST 42* (2012).

[61] Moody, D. The "physics" of notations: toward a scientific basis for constructing visual notations in software engineering. *IEEE Transactions on software engineering 35*, 6 (2009), 756–779.

[62] Musset, J., Juliot, É., Lacrampe, S., Piers, W., Brun, C., Goubet, L., Lussaud, Y., and Allilaire, F. Acceleo user guide. *See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf 2* (2006), 157.

[63] Narayanan, A., Levendovszky, T., Balasubramanian, D., and Karsai, G. Automatic domain model migration to manage metamodel evolution. In *International Conference on Model Driven Engineering Languages and Systems* (2009), Springer, pp. 706–711.

[64] Overbey, J. L., and Johnson, R. E. Generating rewritable abstract syntax trees. In *International Conference on Software Language Engineering* (2008), Springer, pp. 114–133.

[65] Paige, R. F., Matragkas, N., and Rose, L. M. Evolving models in model-driven engineering: State-of-the-art and future challenges. *Journal of Systems and Software 111* (2016), 272–280.

[66] Parr, T., and Fisher, K. Ll(*): The foundation of the antlr parser generator. *ACM Sigplan Notices 46*, 6 (2011), 425–436.

[67] Purdum, J. Beginning c for arduino: Learn c programming for the arduino and compatible microcontroller, 2014.

[68] Richters, M., and Gogolla, M. Ocl: Syntax, semantics, and tools. In *Object Modeling with the OCL*. Springer, 2002, pp. 42–68.

[69] Rohlfshagen, P., Liu, J., Perez-Liebana, D., and Lucas, S. M. Pac-Man Conquers Academia: Two Decades of Research Using a Classic Arcade Game. *IEEE Transactions on Games 10*, 3 (2018), 233–256.

[70] Romano, D., and Pinzger, M. Analyzing the evolution of web services using fine-grained changes. In *2012 IEEE 19th international conference on web services* (2012), IEEE, pp. 392–399.

[71] Roques, A. Plantuml: Open-source tool that uses simple textual descriptions to draw uml diagrams, 2015.

[72] Rose, L., Etien, A., Mendez, D., Kolovos, D., Paige, R., and Polack, F. Comparing model-metamodel and transformation-metamodel coevolution. In *International workshop on models and evolutions* (2010).

[73] Rose, L. M., Paige, R. F., Kolovos, D. S., and Polack, F. A. The epsilon generation language. In *European Conference on Model Driven Architecture-Foundations and Applications* (2008), Springer, pp. 1–16.

[74] Schmidt, D. C. Model-driven engineering. *Computer-IEEE Computer Society- 39*, 2 (2006), 25.

[75] Schmidt, M., and Gloetzner, T. Constructing difference tools for models using the sidiff framework. In *Companion of the 30th international conference on Software engineering* (2008), pp. 947–948.

[76] Smith, R., Koeppe, T., Maurer, J., and Perchik, D. Working draft, standard for programming language c++. *ISO/IEC JTC1/SC22/WG21 document N 4861* (2020).

[77] Smolander, K., Lyytinen, K., Tahvanainen, V.-P., and Marttiin, P. Metaedit—a flexible graphical environment for methodology modelling. In *International Conference on Advanced Information Systems Engineering* (1991), Springer, pp. 168–193.

[78] Soley, R., et al. Model driven architecture. *OMG white paper 308*, 308 (2000), 5.

[79] Sousa, V., Syriani, E., and Fall, K. Operationalizing the integration of user interaction specifications in the synthesis of modeling editors. In *Software Language Engineering* (2019), ACM, pp. 42–54.

[80] Sprinkle, J., and Karsai, G. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing 15*, 3-4 (2004), 291–307.

[81] Steinberg, D., Budinsky, F., Merks, E., and Paternostro, M. *EMF: eclipse modeling framework*. Pearson Education, 2008.

[82] Syriani, E., Luhunu, L., and Sahraoui, H. Systematic mapping study of template-based code generation. *Computer Languages, Systems & Structures 52* (2018), 43–62.

[83] Syriani, E., Vangheluwe, H., Mannadiar, R., Hansen, C., Van Mierlo, S., and Ergin, H. Atompm: A web-based modeling environment. *Demos/Posters/StudentResearch@ MoDELS 2013* (2013), 21–25.

[84] Toulmé, A., and Inc, I. Presentation of emf compare utility. In *Eclipse Modeling Symposium* (2006), pp. 1–8.

[85] van der Storm, T. The rascal language workbench. *CWI. Software Engineering [SEN] 13* (2011), 14.

[86] Van Heesch, D. Doxygen: Source code documentation generator tool. *URL: http://www. doxygen. org* (2008).

[87] Vermolen, S. D., Wachsmuth, G., and Visser, E. Reconstructing complex metamodel evolution. In *International Conference on Software Language Engineering* (2011), Springer, pp. 201–221.

[88] Viyović, V., Maksimović, M., and Perisić, B. Sirius: A rapid development of dsm graphical editor. In *IEEE 18th International Conference on Intelligent Engineering Systems INES 2014* (2014), IEEE, pp. 233–238.

[89] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C., Visser, E., Wachsmuth, G., et al. *DSL engineering: Designing, implementing and using domain-specific languages*. dslbook. org, 2013.

[90] Voelter, M., Benz, S., Dietrich, C., Engelmann, B., Helander, M., Kats, L. C. L., Visser, E., and Wachsmuth, G. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[91] Wachsmuth, G. Metamodel adaptation and model co-adaptation. In *European Conference on Object-Oriented Programming* (2007), Springer, pp. 600–624.

[92] Washizaki, H., Akimoto, M., Hasebe, A., Kubo, A., and Fukazawa, Y. Tcd: a text-based uml class diagram notation and its model converters. In *International Conference on Advanced Software Engineering and Its Applications* (2010), Springer, pp. 296–302.

[93] Wile, D. S. Abstract syntax from concrete syntax. In *Proceedings of the 19th international conference on Software engineering* (1997), pp. 472–480.

Appendix A

Conceptual metamodel

The annotated metamodel of the conceptual aspect of ArduinoDSL

```
@gmf
@namespace(uri="http://www.example.org/arduinoConfiguration", prefix="arduinoConfiguration")
package arduinoConfiguration;

@gmf.diagram
class Project {
  @gmf.label(label.pattern="projectName = {0}")
  attr String projectName;
  val Board board;
}

abstract class Board {
}

@gmf.node(figure="svg", svg.uri="platform:/plugin/ca.iro.umontreal.geodes.arduino.editor.
    configuration/svg/arduinoBoard.svg", size="150,150", label.placement="none", margin="0")
class ArduinoBoard extends Board {
  @gmf.link(label="analogPin")
  val AnalogPin[*] analogPins;
  @gmf.link(label="digitalPin")
  val DigitalPin[*] digitalPins;
  @gmf.link(label="communicationPin")
  val CommunicationPin[*] communicationPins;
}

abstract class Pin {
  @gmf.label(label.pattern="pinNumber = {0}")
  unique attr String pinNumber;
  @gmf.label(label.pattern="pinName = {0}")
  unique attr String pinName;
```

```
30   @gmf.label(label.pattern="pinType = {0}")
31   unique attr PinType type;
32 }
33
34 @gmf.node(label.placement="none", phantom="true")
35 class AnalogPin extends Pin {
36   @gmf.compartment
37   val ArduinoAnalogModule arduinoAnalogModule;
38 }
39
40 @gmf.node(label.placement="none", phantom="true")
41 class DigitalPin extends Pin {
42   @gmf.compartment
43   val ArduinoDigitalModule arduinoDigitalModule;
44 }
45
46 @gmf.node(label.placement="none", phantom="true")
47 class CommunicationPin extends Pin {
48   @gmf.compartment
49   val ArduinoCommunicationModule arduinoCommunicationModule;
50 }
51
52 abstract class Module {
53 }
54
55 abstract class ArduinoModule extends Module {
56   @gmf.label(label.pattern="library = {0}", label.icon="false")
57   unique attr String library;
58 }
59
60 enum PinType {
61   INPUT = 0;
62   OUTPUT = 1;
63 }
64
65 abstract class ArduinoCommunicationModule extends ArduinoModule {
66 }
67
68 abstract class ArduinoDigitalModule extends ArduinoModule {
69 }
70
71 abstract class ArduinoAnalogModule extends ArduinoModule {
72 }
73
```

```
74 @gmf.node(figure="svg", svg.uri="platform:/plugin/ca.iro.umontreal.geodes.arduino.editor.
       configuration/svg/rfid.svg",
75 size="150,150", label.placement="external", label="cards", label.pattern="cards = {0}", margin="
       0")
76 class RFID extends ArduinoDigitalModule {
77   attr String[*] cards;
78 }
79
80 @gmf.node(figure="svg", svg.uri="platform:/plugin/ca.iro.umontreal.geodes.arduino.editor.
       configuration/svg/blueLed.svg",
81 size="150,150", label.placement="none", margin="0")
82 class LED extends ArduinoDigitalModule {
83   @gmf.label(label.pattern="brightness = {0}")
84   attr int brightness;
85 }
```

Appendix B

Behavioral metamodel

The Xtext grammar representing the behavioral metamodel of ArduinoDSL

```
grammar ca.iro.umontreal.geodes.arduino.editor.interaction.FixedGrammar with ca.iro.umontreal.geodes.arduino.editor.interaction.DynamicGrammar
generate fixedGrammar "http://www.iro.ca/umontreal/geodes/arduino/editor/interaction/FixedGrammar"
Project:
  project += InfiniteLoop | Setup;

InfiniteLoop:
  infiniteLoop += Loop | CustomIf | CustomCode | Declaration | ArduinoFunctions | Comments | CustomAssignment | FunctionDeclaration | FunctionCall | RFID125Functions;

RFID125Functions: BranchRFID125 | ReadCodeRFID125 | CheckCardRegistrationRFID125 | WriteCodesRFID125 | ClearCodesRFID125;

CheckCardRegistrationRFID125: 'RFID125' 'checkRegistration' card+=STRING 'on' pin=Pins;

ClearCodesRFID125: 'RFID125' 'clearCards' 'on' pin=Pins;

WriteCodesRFID125: 'RFID125' 'registerCards' codes+=STRING 'on' pin=Pins;

ReadCodeRFID125: 'RFID125' 'readCard' 'on' pin=Pins ('->' (variableType='String')? variableName+=ID)?;

FunctionDeclaration:
  'FUNCTION' functionName=ID '{'
    (functionBody+=InfiniteLoop)+
  '}';

FunctionCall: 'CALL' functionName=ID;

```

```
26 Comments: 'BEGIN:COMMENTS' comments+=STRING 'END:COMMENTS';
27
28 Setup:
29   'SETUP' '{'
30     (setup+=InfiniteLoop)+
31   '}';
32
33 ArduinoFunctions: DigitalFunctions| AnalogFunctions | AdvancedFunctions | TimeFunctions |
       SerialFunctions;
34
35 SerialFunctions: SerialPrint | SerialBegin | SerialRead | SerialAvailable;
36
37 SerialRead: 'SERIAL' 'read' ('->' (variableType='string')? variableName=ID)?;
38
39 SerialAvailable: 'SERIAL' 'available' ('->' (variableType='int')? variableName=ID)?;
40
41 SerialPrint: 'SERIAL' 'print' message+=STRING;
42
43 SerialBegin:
44   'SERIAL' 'begin' speed=INT ('withConfig' config=('SERIAL_5N1' | 'SERIAL_6N1' | 'SERIAL_7N1' |
         'SERIAL_8N1' | 'SERIAL_5N2' | 'SERIAL_6N2' | 'SERIAL_7N2' | 'SERIAL_8N2' | 'SERIAL_5E1' |
         'SERIAL_6E1' | 'SERIAL_7E1' | 'SERIAL_8E1' | 'SERIAL_5E2' | 'SERIAL_6E2' | 'SERIAL_7E2' |
         'SERIAL_8E2' | 'SERIAL_5O1' | 'SERIAL_6O1' | 'SERIAL_7O1' | 'SERIAL_8O1' | 'SERIAL_5O2' |
         'SERIAL_6O2' | 'SERIAL_7O2' | 'SERIAL_8O2'))?
45 ;
46
47 AdvancedFunctions:
48   ShiftIn | PulseIn | PulseInLong | NoTone | ShiftOut | Tone
49 ;
50
51 ShiftOut:
52   'SHIFT' 'out' value=INT 'from' clockPin=Pins 'to' dataPin=Pins 'inOrder' order=('MSBFIRST' | '
         LSBFIRST')
53 ;
54
55 ShiftIn:
56   'SHIFT' 'in' 'from' clockPin=Pins 'to' dataPin=Pins 'inOrder' order=('MSBFIRST' | 'LSBFIRST')
         ('->' (variableType='byte')? variableName=ID)?
57 ;
58
59 NoTone:
60   'TONE' 'stop' 'on' pin=Pins
61 ;
62
63 Tone:
```

```
  'TONE' 'play' frequency=INT ('during' time=INT)? 'on' pin=Pins
;

PulseIn:
  'PULSE' 'read' pulseType=('HIGH' | 'LOW') 'on' pin=Pins ('STOP_IF_NO_PULSE' 'in' time=INT)? ('->' (variableType='long')? variableName=ID)?
;

PulseInLong:
  'PULSE' 'readLong' pulseType=('HIGH' | 'LOW') 'on' pin=Pins ('STOP_IF_NO_PULSE' 'in' time=INT) ? ('->' (variableType='long')? variableName=ID)?
;


TimeFunctions:
  Micros | Millis | Delay | DelayMicroseconds
;

Millis:
  time+='GET_RUNNING_TIME_IN_MILLISECONDS' ('->' (variableType='long')? variableName=ID)?
;

Micros:
  time+='GET_RUNNING_TIME_IN_MICROSECONDS' ('->' (variableType='long')? variableName=ID)?
;

DelayMicroseconds:
  'WAIT' time=INT 'MICROSECONDS';

Delay: 'WAIT' time=INT 'MILLISECONDS';

AnalogFunctions: AnalogRead | AnalogWrite | AnalogReference;

AnalogReference: 'ANALOG' 'reference' voltage=('DEFAULT' | 'INTERNAL' | 'INTERNAL1V1' | 'INTERNAL2V56' | 'EXTERNAL');

AnalogRead: 'ANALOG' 'read' pin=Pins ('->' (variableType='int')? variableName=ID)?;

AnalogWrite: 'ANALOG' 'write' (intValue=INT | idValue=ID) 'on' pin=Pins;

DigitalFunctions: DigitalRead | DigitalWrite | PinMode;

DigitalRead: 'DIGITAL' 'read' pin=Pins ('->' (variableType='int')? variableName=ID)?;

DigitalWrite: 'DIGITAL' 'write' value=('HIGH' | 'LOW') 'on' pin=Pins;
```

```
106
107 PinMode: 'MODE' 'set' mode=('INPUT' | 'OUTPUT' | 'INPUT_PULLUP') 'on' pin=Pins;
108
109 CustomCode: customCode+=STRING;
110
111 Loop: CustomFor | CustomWhile | CustomDoWhile;
112
113 CustomFor:
114   'BEGINNING_FROM' begin=INT 'TO' end=INT 'JUMPING_BY' step=INT 'REPEAT:'
115     (action+=Action)+
116   'END:REPEAT';
117
118 CustomWhile:
119   'WHILE' whileCondition=CustomCode 'DO:'
120     (whileAction+=Action)+
121   'END:WHILE';
122
123 CustomDoWhile:
124   'DO:'
125     (dowhileAction+=Action)+
126   'WHILE' dowhileCondition=CustomCode
127   'END:DOWHILE';
128
129 CustomCondition: CustomElif | CustomElse;
130
131 CustomIf:
132   'IF' ifCondition=CustomCode
133     (ifAction+=Action)+
134     (ifElse+=CustomCondition)*
135   'END:IF';
136
137 Action: action+=InfiniteLoop;
138
139 CustomElif:
140   'ELSEIF' elifCondition+=CustomCode
141     (elifAction+=Action)+;
142
143 CustomElse:
144   'ELSE'
145     (elseAction+=Action)+;
146
147 CustomDeclaration: (variableType=VariableType) (nom=ID) ('=' assignmentValues+=AssignmentValues)
        ?;
148
149 AssignmentValues: valeur+=Values (operateur+=('+' | '-' | '*' | '/') operationValues+=Values)*;
```

```
150
151 CustomAssignment: nom=ID '=' assignmentValues+=AssignmentValues;
152
153 VariableType: variableType=('String' | 'int' | 'char' | 'bool');
154
155 FunctionType: functionType=('String' | 'int' | 'char' | 'bool' | 'void');
156
157 Values: stringValue=STRING | intValue+=INT | constantValue=('HIGH' | 'LOW' | 'true' | 'false') |
        variableValue=ID;
```

Appendix C

The board code generator

The Arduino code generator for the ArduinoDSL board model

```
1 // libraries imported here
2 \#include <SoftwareSerial.h> // Serial library
3 [% for(aModule in Board!Module) {
4 if(aModule.library.isDefined()) { %]
5 \#include "[%=aModule.library%].h"
6 [% }
7 } %]
8
9 // pins defined here
10 [% for(aPin in Board!Pin) {
11 if(aPin.pinNumber.isDefined() and aPin.pinName.isDefined()) { %]
12 const int [%=aPin.pinName%] = [%=aPin.pinNumber%];
13 [% }
14 } %]
15
16 // initial configuration defined here
17 // will run just once
18 void setup() {
19 [% for(aPin in Board!Pin) {
20 if(aPin.pinName.isDefined()){
21 if (aPin.type == "OUTPUT") { %]
22 pinMode([%=aPin.pinName%], [%=aPin.type%]);
23 [% }
24 else { %]
25 pinMode([%=aPin.pinName%], INPUT);
26 [% }
27 }
28 else if(aPin.pinNumber.isDefined()){
29 if (aPin.type == "OUTPUT") { %]
30 pinMode([%=aPin.pinNumber%], [%=aPin.type%]);
```

```
31 [% }
32 else { %]
33 pinMode([%=aPin.pinNumber%], INPUT);
34 [% }
35 }
36 } %]
```

On line 2, we import the SoftwareSerial library[1] which allows serial communication on the digital pins and not only on the UART pin. On lines 3 to 7, we import the libraries defined in the board model. On lines 10 to 15, we declare and assign names to the pins used in the board model, if it was defined. To complete the configuration of the modules, we generate their mode (either INPUT or OUTPUT), on lines 19 to 37.

[1]SoftwareSerial library: `https://www.arduino.cc/en/Reference/softwareSerial`

Appendix D

The sketch code generator

The Arduino code generator for the ArduinoDSL sketch model

```
[% // find the setup if it exists
for (aSetup in Sketch!Setup) {
for(anInstance in aSetup.setup) { %]
[%=anInstance.recursive_op()%]
[% }
} %]
}

// the behavior of an infinite loop
// will always run unless a reboot is done
void loop()
{
[% for (aProject in Sketch!Project) {
for(anInstance in aProject.project) {
if(not anInstance.isTypeOf(Sketch!InfiniteLoop)) {%]
[%=anInstance.recursive_op()%]
[% }
else { %]
[%=anInstance.infiniteLoop.first().recursive_op()%]
[% }
}
} %]
}


[*    Operation template *]


[% operation Sketch!Comments recursive_op() { %]
/*
```

```
[%=self.comments.first()%]
*/
[% } %]

[% operation Sketch!CheckCardRegistrationRFID recursive_op() {
var fonction = '' + self.pin.pins.first() + '.testerCode("' + self.card.first() + '");';%]
[%=fonction%]
[% } %]

[% operation Sketch!ClearCodesRFID recursive_op() {
var fonction = '' + self.pin.pins.first() + '.effacerCodes();';%]
[%=fonction%]
[% } %]

[% operation Sketch!WriteCodesRFID recursive_op() {
var fonction = '' + self.pin.pins.first() + '.ecrireCodes("' + self.codes.first() + '");';%]
[%=fonction%]
[% } %]

[% operation Sketch!ReadCodeRFID recursive_op() {
var fonction = '';
if(self.variableName.first().isDefined()){
if(self.variableType.isDefined()){
fonction = fonction + self.variableType + ' ' + self.variableName.first() + ' = ' + self.pin.
    pins.first() + '.lireCode();';
}
else {
fonction = fonction + self.variableName.first() + ' = ' + self.pin.pins.first() + '.lireCode();'
    ;
}
}
else{
fonction = fonction + self.pin.pins.first() + '.lireCode();';
}%]
[%=fonction%]
[% } %]

[% operation Sketch!SerialInfosRFID recursive_op() {
var fonction = '';
if(self.state='ON') {
fonction = fonction + self.pin.pins.first() + '.activerSerialInfos();';
}
else{
fonction = fonction + self.pin.pins.first() + '.desactiverSerialInfos();';
}%]
```

```
74 [%=fonction%]
75 [% } %]
76
77 [% operation Sketch!BeginLCD recursive_op() {
78 var fonction = '' + self.pin.pins.first() + '.begin(' + self.columns + ',' + self.rows + ');';%]
79 [%=fonction%]
80 [% } %]
81
82
83 [% operation Sketch!PrintLCD recursive_op() {
84 var fonction = '' + self.pin.pins.first() + '.print("' + self.message.first() + '");';%]
85 [%=fonction%]
86 [% } %]
87
88
89 [% operation Sketch!SetCursorLCD recursive_op() {
90 var fonction = '' + self.pin.pins.first() + '.setCursor(' + self.column + ',' + self.row + ');'
       ;%]
91 [%=fonction%]
92 [% } %]
93
94
95 [% operation Sketch!SetRGBLCD recursive_op() {
96 var blue = 0;
97 var red = 0;
98 var green = 0;
99 if(self.blue.isDefined()) {
100 blue = self.blue;
101 }
102 if(self.red.isDefined()) {
103 red = self.red;
104 }
105 if(self.green.isDefined()) {
106 green = self.green;
107 }
108 var fonction = '' + self.pin.pins.first() + '.setRGB(' + red + ',' + green + ',' + blue + ');'
       ;%]
109 [%=fonction%]
110 [% } %]
111
112
113 [% operation Sketch!SerialRead recursive_op() {
114 if(self.variableName.first().isDefined()) {
115 if(self.variableType.isDefined()) {%]
116 String [%=self.variableName.first()%] = Serial.readString();
```

```
117 [% } else {%]
118 [%=self.variableName.first()%] = Serial.readString();
119 [%}
120 }
121 else {%]
122 Serial.readString();
123 [% }
124 } %]
125
126
127 [% operation Sketch!SerialAvailable recursive_op() {
128 if(self.variableName.first().isDefined()) {
129 if(self.variableType.isDefined()) {%]
130 int [%=self.variableName.first()%] = Serial.available();
131 [% }
132 else {%]
133 [%=self.variableName.first()%] = Serial.available();
134 [%}
135 }
136 else {%]
137 Serial.available();
138 [% }
139 } %]
140
141
142 [% operation Sketch!SerialPrint recursive_op() { %]
143 Serial.println("[%=self.message.first()%]");
144 [% } %]
145
146
147 [% operation Sketch!SerialBegin recursive_op() {
148 if(self.config.isDefined()) { %]
149 Serial.begin([%=self.speed%], [%=self.config%]);
150 [% }
151 else { %]
152 Serial.begin([%=self.speed%]);
153 [% }
154 } %]
155
156
157 [% operation Sketch!ShiftOut recursive_op() { %]
158 shiftOut([%=self.dataPin.pins.first()%], [%=self.clockPin.pins.first()%], [%=self.order%], [%=
        self.value%]);
159 [% } %]
160
```

```
161
162 [% operation Sketch!ShiftIn recursive_op() {
163 if(self.variableName.first().isDefined()) {
164 if(self.variableType.first().isDefined()) {%]
165 byte [%=self.variableName.first()%] = shiftIn([%=self.dataPin.pins.first()%], [%=self.clockPin.
        pins.first()%], [%=self.order%]);
166 [% }
167 else {%]
168 [%=self.variableName.first()%] = shiftIn([%=self.dataPin.pins.first()%], [%=self.clockPin.pins.
        first()%], [%=self.order%]);
169 [%}
170 }
171 else {%]
172 shiftIn([%=self.dataPin.pins.first()%], [%=self.clockPin.pins.first()%], [%=self.order%]);
173 [% }
174 } %]
175
176
177 [% operation Sketch!NoTone recursive_op() { %]
178 noTone([%=self.pin.pins.first()%]);
179 [% } %]
180
181
182 [% operation Sketch!Tone recursive_op() {
183 if(self.time > 0) {%]
184 tone([%=self.pin.pins.first()%], [%=self.frequency%], [%=self.time%]);
185 [% }
186 else{%]
187 tone([%=self.pin.pins.first()%], [%=self.frequency%]);
188 [%}
189 } %]
190
191
192 [% operation Sketch!PulseIn recursive_op() {
193 if(self.time > 0) {
194 if(self.variableName.first().isDefined()) {
195 if(self.variableType.first().isDefined()) {%]
196 unsigned long [%=self.variableName.first()%] = pulseIn([%=self.pin.pins.first()%], [%=self.
        pulseType%], [%=self.time%]);
197 [% }
198 else {%]
199 [%=self.variableName.first()%] = pulseIn([%=self.pin.pins.first()%], [%=self.pulseType%], [%=
        self.time%]);
200 [%}
201 }
```

```
202 else {%]
203 pulseIn([%=self.pin.pins.first()%], [%=self.pulseType%], [%=self.time%]);
204 [%}
205 }
206 else {
207 if(self.variableName.first().isDefined()) {
208 if(self.variableType.first().isDefined()) { %]
209 unsigned long [%=self.variableName.first()%] = pulseIn([%=self.pin.pins.first()%], [%=self.
        pulseType%]);
210 [% }
211 else {%]
212 [%=self.variableName.first()%] = pulseIn([%=self.pin.pins.first()%], [%=self.pulseType%]);
213 [%}
214 }
215 else {%]
216 pulseIn([%=self.pin.pins.first()%], [%=self.pulseType%]);
217 [%}
218 }
219 } %]
220
221
222 [% operation Sketch!PulseInLong recursive_op() {
223 if(self.time > 0) {
224 if(self.variableName.first().isDefined()) {
225 if(self.variableType.first().isDefined()) {%]
226 unsigned long [%=self.variableName.first()%] = pulseInLong([%=self.pin.pins.first()%], [%=self.
        pulseType%], [%=self.time%]);
227 [% }
228 else {%]
229 [%=self.variableName.first()%] = pulseInLong([%=self.pin.pins.first()%], [%=self.type%], [%=self
        .time%]);
230 [%}
231 }
232 else {%]
233 pulseInLong([%=self.pin.pins.first()%], [%=self.pulseType%], [%=self.time%]);
234 [%}
235 }
236 else {
237 if(self.variableName.first().isDefined()) {
238 if(self.variableType.first().isDefined()) {%]
239 unsigned long [%=self.variableName.first()%] = pulseInLong([%=self.pin.pins.first()%], [%=self.
        pulseType%]);
240 [% }
241 else {%]
242 [%=self.variableName.first()%] = pulseInLong([%=self.pin.pins.first()%], [%=self.pulseType%]);
```

```
[%}
}
else {%]
pulseInLong([%=self.pin.pins.first()%], [%=self.pulseType%]);
[%}
}
} %]


[% operation Sketch!Millis recursive_op() {
if(self.variableName.first().isDefined()) {
if(self.variableType.first().isDefined()) {%]
unsigned long [%=self.variableName.first()%] = millis();
[% }
else {%]
[%=self.variableName.first()%] = millis();
[%}
}
else {%]
millis();
[%}
} %]


[% operation Sketch!Micros recursive_op() {
if(self.variableName.first().isDefined()) {
if(self.variableType.first().isDefined()) {%]
unsigned long [%=self.variableName.first()%] = micros();
[% }
else {%]
[%=self.variableName.first()%] = micros();
[%}
}
else {%]
micros();
[% }
} %]


[% operation Sketch!Delay recursive_op() { %]
delay([%=self.time%]);
[% } %]


[% operation Sketch!DelayMicroseconds recursive_op() { %]
```

```
288 delayMicroseconds([%=self.time%]);
289 [% } %]
290
291
292 [% operation Sketch!AnalogReference recursive_op() { %]
293 analogReference([%=self.voltage%]);
294 [% } %]
295
296
297 [% operation Sketch!AnalogRead recursive_op() {
298 if(self.variableName.first().isDefined()){
299 if(self.variableType.first().isDefined()){%]
300 int [%=self.variableName.first()%] = analogRead([%=self.pin.pins.first()%]);
301 [% }
302 else{%]
303 [%=self.variableName.first()%] = analogRead([%=self.pin.pins.first()%]);
304 [%}
305 }
306 else{%]
307 analogRead([%=self.pin.pins.first()%]);
308 [%}
309 } %]
310
311
312 [% operation Sketch!AnalogWrite recursive_op() {
313 if(self.idValue.first().isDefined()) {%]
314 analogWrite([%=self.pin.pins.first()%], [%=self.idValue.first()%]);
315 [% }
316 else {%]
317 analogWrite([%=self.pin.pins.first()%], [%=self.intValue.first()%]);
318 [%}
319 } %]
320
321 [% operation Sketch!DigitalRead recursive_op() {
322 if(self.variableName.first().isDefined()){
323 if(self.variableType.first().isDefined()){%]
324 int [%=self.variableName.first()%] = digitalRead([%=self.pin.pins.first()%]);
325 [% }
326 else{%]
327 [%=self.variableName.first()%] = digitalRead([%=self.pin.pins.first()%]);
328 [%}
329 }
330 else{%]
331 digitalRead([%=self.pin.pins.first()%]);
332 [%}
```

```
} %]


[% operation Sketch!DigitalWrite recursive_op() { %]
digitalWrite([%=self.pin.pins.first()%], [%=self.value%]);
[% } %]


[% operation Sketch!PinMode recursive_op() { %]
pinMode([%=self.pin.pins.first()%], [%=self.mode%]);
[% } %]


[%
operation Sketch!CustomCode recursive_op() { %]
[%=self.customCode.first()%]
[% }
%]


[% operation Sketch!CustomFor recursive_op() { %]
for(int i = [%=self.begin.first()%]; i < [%=self.end.first()%]; i+=[%=self.step.first()%]) {
[%for(anAction in self.action) {%]
[%=anAction.action.first().recursive_op()%]
[% } %]
}
[% } %]


[% operation Sketch!CustomWhile recursive_op() { %]
while([%=self.whileCondition.customCode.first().first()%]) {
[%for(anAction in self.whileAction) {%]
[%=anAction.action.first().recursive_op()%]
[% } %]
}
[% } %]


[% operation Sketch!CustomDoWhile recursive_op() { %]
do {
[%for(anAction in self.dowhileAction) {%]
[%=anAction.action.first().recursive_op()%]
[% } %]
} while([%=self.dowhileCondition.customCode.first().first()%]);
[% } %]
```

```
378
379
380 [% operation Sketch!CustomIF recursive_op() { %]
381 if ([%=self.ifCondition.customCode.first().first()%]) {
382 [%for(anAction in self.ifAction) {%]
383 [%=anAction.action.first().recursive_op()%]
384 [% } %]
385 }
386 [% if(not self.ifElse.isEmpty()) {
387 for(anAlternative in self.ifElse) {%]
388 [%=anAlternative.recursive_op()%]
389 [% }
390 }
391 } %]
392
393
394 [% operation Sketch!CustomElif recursive_op() { %]
395 else if ([%=self.elifCondition.customCode.first().first()%]) {
396 [%for(anAction in self.elifAction) {%]
397 [%=anAction.action.first().recursive_op()%]
398 [% } %]
399 }
400 [% } %]
401
402
403 [% operation Sketch!CustomElse recursive_op() { %]
404 else {
405 [%for(anAction in self.elseAction) {%]
406 [%=anAction.action.first().recursive_op()%]
407 [% } %]
408 }
409 [% } %]
410
411
412 [% operation Sketch!CustomDeclaration recursive_op() {
413 if(not (self.assignmentValues.size() > 0)) { %]
414 [%=self.variableType.first().variableType.first()%] [%=self.nom.first()%];
415 [% }
416 else { %]
417 [%=self.variableType.first().variableType.first()%] [%=self.nom.first()%] = [%=self.
        assignmentValues.valeur.first().first().getOperationValues()%][%=(self.getAssignmentValues()
        )%];
418 [% }
419 } %]
420 [*if(not (self.assignmentValues.operationValues.size() > 0)) { %]
```

```
[%=self.variableType.first().variableType.first()%] [%=self.nom.first()%] = [%=self.
    assignmentValues.valeur.first().first().getOperationValues()%];
[% }
else { %]
[%=self.variableType.first().variableType.first()%] [%=self.nom.first()%] = [%=self.
    assignmentValues.valeur.first().first().getOperationValues()%] [%=(self.getAssignmentValues
    ())%];
[% }
}
} %*]

[% operation Sketch!FunctionCall recursive_op() {%]
[%=self.functionName.first()%]();
[% } %]

[% operation Sketch!FunctionDeclaration recursive_op() {%]
void [%=self.functionName.first()%](){
[%for(aFonctionBody in self.fonctionBody) {%]
[%=aFonctionBody.fonctionBody.first().recursive_op()%]
[% } %]
}
[% } %]


[% operation Sketch!CustomAssignment recursive_op() { %]
[%=self.nom.first()%] = [%=(self.assignmentValues.valeur.first().first().getOperationValues() +
    self.getAssignmentValues())%];
[% } %]


[% operation Any getAssignmentValues() {
var operations = '';
if(self.assignmentValues.first().operateur.size() > 0) {
var operators = self.assignmentValues.first().operateur;
var values = self.assignmentValues.first().operationValues;
for(count in Sequence{0..(values.size()-1)}){
operations = operations + ' ' + operators.at(count) + ' ' + values.at(count).getOperationValues
    ();
}
}
return operations;
} %]


[% operation Any getOperationValues() {
```

```
var operations;
if(self.constantValue.first().isDefined()) {
operations = self.constantValue.first();
}
else if(self.stringValue.first().isDefined()) {
operations = self.stringValue.first();
}
else if(self.intValue.first().isDefined()) {
operations = self.intValue.first();
}
else if(self.functionValue.first().isDefined()) {
operations = self.functionValue.first().recursive_op();
}
else if(self.variableValue.first().isDefined()) {
operations = self.variableValue.first();
}
return operations;
} %]


[% operation Sketch!InfiniteLoop recursive_op() {
self.infiniteLoop.first().recursive_op();
}
%]
```

Appendix E

Helper class

The Helper class which transforms a sketch to an XMI file

```
1  package ca.iro.umontreal.geodes.arduino.editor.interaction.generator;
2
3  import java.io.IOException;
4  import java.util.HashMap;
5  import java.util.Map;
6  import org.eclipse.emf.common.util.URI;
7  import org.eclipse.emf.ecore.resource.Resource;
8  import org.eclipse.emf.ecore.xmi.impl.XMIResourceImpl;
9
10 public class Helper {
11   public static void saveResourceAsXmi(Resource resource) {
12     try {
13       Map<String, String> saveOptions = new HashMap<String, String>();
14       Resource xmiResource = new XMIResourceImpl(URI.createURI(resource.getURI().toString().
             replace("sketch", "xmi")));
15       System.out.println(xmiResource);
16       xmiResource.getContents().add(resource.getContents().get(0));
17       saveOptions.put(org.eclipse.emf.ecore.xmi.XMIResource.OPTION_ENCODING,"UTF-8");
18       xmiResource.save(saveOptions);
19       System.out.println("XMI sketch file created.");
20     } catch (IOException e) {
21       System.out.println("Error during creation of XMI.");
22       e.printStackTrace();
23     }
24   }
25 }
```

Appendix F

Pac-Man game sketch model

The sketch model of the Pac-Man game editor in ArduinoDSL.

```
SETUP {
  SERIAL begin 9600
  int oldPotentioValue = 0
  MOUSE enable
}
FUNCTION scream {
  SERIAL available -> int freeSerial
  IF 'freeSerial > 0'
    SERIAL read -> String message
  IF 'message=="scream"'
    TONE play 1000 on buzzerPin
    TIME waitMilliseconds 500
    TONE stop on buzzerPin
  END:IF
  END:IF
}
DIGITAL read on pacmanButton -> pacmanPressed
DIGITAL read on ghostButton -> ghostPressed
DIGITAL read on foodButton -> foodPressed
DIGITAL read on gridButton -> gridPressed
IF 'pacmanPressed'
  SERIAL print "addPacman"
  TIME waitMilliseconds 500
ELSEIF 'ghostPressed'
  SERIAL print "addGhost"
  TIME waitMilliseconds 500
ELSEIF 'foodPressed'
  SERIAL print "addFood"
  TIME waitMilliseconds 500
ELSEIF 'gridPressed'
```

```
  SERIAL print "addGrid"
  TIME waitMilliseconds 500
END:IF
MOUSE readAxis on A1 -> int xValue
MOUSE readAxis on A0 -> int yValue
MOUSE move xValue and yValue and 0
MOUSE pressed "MOUSE_LEFT" -> mousePressed
IF 'mousePressed'
  SERIAL print "selectElement"
  TIME waitMilliseconds 500
  WHILE 'mousePressed'
  END:WHILE
  SERIAL print "moveElement"
  TIME waitMilliseconds 500
END:IF
ANALOG read lightSensorPin -> int brightness
IF 'brightness < 600'
  SERIAL print "addFood"
  CALL scream
  TIME waitMilliseconds 3000
END:IF
CALL scream
RFID125 readCard on rfidSensor -> String code
IF 'code=="7871946"'
  SERIAL print "SimplePacMan"
  TIME waitMilliseconds 2000
ELSEIF 'code=="7878677"'
  SERIAL print "3DPacMan"
  TIME waitMilliseconds 2000
END:IF
ANALOG read potentioPin -> int potentioValue
MATH map potentioValue and 0 and 1023 and 1 and 10 -> potentioValue
IF 'potentioValue != oldPotentioValue'
  oldPotentioValue = potentioValue
  String zoom = "zoom" + potentioValue
  SERIAL print zoom
  TIME waitMilliseconds 500
END:IF
```