# Automated Code Generation for NFC-based Access Control

Ludovico Iovino
*Computer Science Department*
*Gran Sasso Science Institute*
L'Aquila, Italy
ludovico.iovino@gssi.it

Martina De Sanctis
*Computer Science Department*
*Gran Sasso Science Institute*
L'Aquila, Italy
martina.desanctis@gssi.it

Maria Teresa Rossi
*Computer Science Department*
*Gran Sasso Science Institute*
L'Aquila, Italy
mariateresa.rossi@gssi.it

***Abstract*—Access control systems provide a security application for controlling access of persons through controlled gates. The gate, such as a door, can have a lock mechanism for securing the area from unauthorized access. Radio Frequency Identification (RFID) / Near Field Communication (NFC) is a key technology for sensor communication enabling a ubiquitous and pervasive computing network, popularly employed in access control systems. In this paper we investigate a case study about the migration of an old chip card based access control system to this new way of authorizing people, based on NFC tags. To this aim, we propose a Model Driven approach that supports (semi) automatic code generation for enabling the communication between an IoT infrastructure and platforms for Facility Access Management. Specifically, the approach combines the benefits of NFC and Tinkerforge (i.e., an open source hardware platform) technologies with model driven techniques. As a consequence, it allows the reduction of the implementation efforts, by also providing snippets as exemplary code usage, while making the whole development process less error-prone and time consuming.**

## I. Introduction

Access control systems provide a security application for supervising access of persons through controlled gates. The gate, such as a door, can have a lock mechanism for securing the area from unauthorized access. Internet of Things (IoT) is a new concept of integrated network of devices in the field of information technology where devices communicate with each other. IoT security in building access control is a relevant field where building security is implemented through IoT network. Every device is connected with the related sensors, which trigger the control mechanism communicating with the rest of the infrastructure.

Model-driven engineering [1] aims at tackling the complexity of software systems via abstraction, using models as first-class entities. Models are employed both for descriptive and prescriptive purposes [2]. Model Driven Development (MDD) approaches open to code-generation strategies, to reduce time to market and improve quality of software [3]. Furthermore, Model Driven approaches, and in particular model transformations, can be successfully employed for supporting platform independent to platform specific translation, also implementing multi-platform code generation from single input model of the system [4], [5]. This is quite relevant, given that nowadays cross-platform development is a barrier for solution providers due to the high cost of development and maintenance of targeting development to different platforms.

This paper explores a case study about the migration of an old chip card based access control system to a new way of authorizing people with Near Field Communication (NFC) tags. In the chip card based version of the system, the standard plastic card which contains an embedded microchip has to be physically inserted in a reader to be read; in case data stored on the chip allows the access, it enables the electric door lock. Usually, the card reader hosts the logic for the access control and, for each equipped door, the code implementing this logic has to be distributed and released on the reader(s). In addition to the access control system, a human resource management system is used to associate a person to the card, to have a registry in case the card gets lost or a check is needed.

Currently, a new way of interaction is provided by NFC technology [6], which enables the 'touching paradigm' where the interaction can be identified as "the deliberate bringing together of two devices, for the purpose of obtaining services" [7]. NFC requires two compatible devices to be close to each other, basically touching them for enabling contactless identification and interaction. Moreover, most of the commercial solutions for access control management also offer pre-packaged software systems for human resources management. The main issue with these existing solutions is the impossibility of customizing the authorization logic, provided by the tool vendor. In case they offer customization, this service is extra-paid and quite expensive.

Tinkerforge [8] is an open source affordable system of building blocks using the concept of pluggable module, helping to simplify the implementation of IoT projects. The implementation of the building blocks is based on intuitive API available for many programming languages, e.g., Java, PHP, Python. This system, offering a high degree of abstraction, might ease the technical implementation with less code to be developed with respect to other components. The main advantage in selecting a target technology such as Tinkerforge is that it can be used to build systems interacting with other platforms and customize the access control algorithm for the required purpose, since the code has to be developed. Moreover, the building blocks of the infrastructure can be selected and mounted, based on the requirements, and easily extended. Despite this selection offers a good trade-off between development and provided functionalities, the code controlling the devices has to be manually written, although some examples and documented APIs are provided on the website.

In this paper, we propose a Model Driven approach that supports (semi) automatic code generation for enabling the communication between an IoT infrastructure and an existing web platform for Facility Access Management. In particular, by combining the benefits of NFC and Tinkerforge technologies together with model driven techniques, the presented approach is able to generate software components, in the selected target language. This way, it allows reducing the implementation effort, also by offering snippets as exemplary code usage, while making the whole development process less error-prone and time consuming.

The paper is organized as follows: section II offers a case study for introducing the domain in which the proposed approach, presented in section III, is applied. Section IV explores the related works and section V draws some conclusions, by also briefly mentioning possible future developments.

## II. Case study

In this section, we expose the case study we designed from a project experience with a company. Specifically, the company we deal with it is a gym that is organized in diverse rooms, distributed in several floors of a building. Each room is equipped with different facilities to host specific activities (e.g., training, rehab, pilates). Activities are scheduled in different time slots from Monday to Saturday, with diverse frequencies, and assigned to the available rooms. The clients of the gym can enroll in different courses and/or activities, by choosing different offers and subscriptions. The company wants to exploit the subscriptions that clients hold, to automatically check if they have access or not to each specific room in the building. This way, it can be easily controlled if only authorized clients regularly registered in certain activities, effectively attend it.

We use a metamodeling approach to represent the case study introduced above. To better comprehend the application domain, we engineered the requirements for the system in the metamodel in Fig 1. An *AccessControlSystem* is organized to manage a list of *Activities* that can be conducted in the facility, and specifically in the available *Room*s in specific *TimeSlot*s. In fact, each room has a *Timetable* of activities, performed in specific *WEEKDAY*s and time. The system can register *Client*s associated to a *Person* with all the fields for the identification (hidden for lack of space). Each client can be linked to *MemberShip*s with a specific *start* and *end* date related to an *Activity*. The access control system can manage who is authorized to access a room in a specific date and time. This can be derived by retrieving all the activities of a client and by matching them with the timetable of the activities. The authorization and identification of the client then is delegated to the access control infrastructure, that in this case is implemented with the selected NFC technology. This technology mounted near to the door of a room will identify the tag used by the client and it will trigger the authorization mechanism, by also communicating with the access control system. For instance, the access control system can expose web services or API devoted to this task.

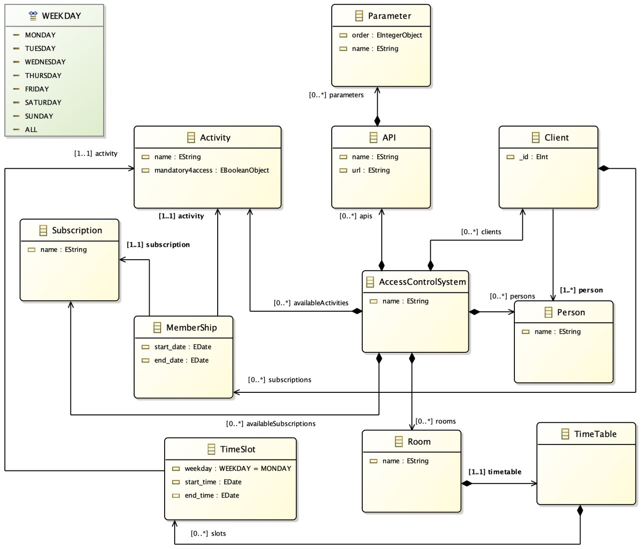


Fig. 1. Access Control Metamodel.

**Case Study Configuration.** As introduced in section I, Tinkerforge is an open source platform of building blocks belonging to three categories, namely *Bricks* that, in turn, can control different modules called *Bricklets*. Then, the communication interface of these building blocks can be extended by using *Master Extensions*. More precisely, each Brick has one task, which in our case study consists in checking the communication and authorization. Bricks are stackable, in the sense that can be mounted on top of each other by creating a hierarchy. For simplicity, in this paper we consider non-stackable Bricks. Bricklets, can be connected to a Brick with a cable and depending on the Bricklet type, different sensors, displays, I/O interfaces can be added. Master extensions are used to change the interface from USB to either Ethernet or WIFI.

To support a basic configuration in our case study, we consider an architecture composed of a single Brick per door (room) and multiple connected Bricklets, namely an NFC reader, a speaker (beeper), a led display and a relay to control the door lock. Relays are switches that can electronically open and close circuits and, in our case study, they control 4 circuits (from which the name of the component quad relay). Moreover, each client has to receive a NFC tag (card, sticker or bracelet) that has to be enabled for the entrance. Logically, the steps to be performed when entering a room are the following: the NFC reader receives the NFC tag identifier and it checks the authorization with the access control system. In the case in which the authorization is provided, the NFC reader emits a beep by using the speaker, it shows an authorization message on the display, and it enables one of the channels of the relay controlling the door lock. If the authorization is denied, instead, the NFC reader emits a sound and displays the error message. All these operations can be implemented by using the Tinkerforge API.

In the following, we show the Java code that must be manually written, in the absence of automatic code generation, to implement the explained control logic among the various sensors and bricklets, for our case study. In listing 1 we report the Java code developed for the logic of a single brick. This initialized Brick has UID (identifier) `hr4tx` and will be powered by PoE (power over ethernet) extension, with ip address `192.168.1.2` (localhost in our LAN) for communication.

```
1  package tinkerForgeGen;
2
3  import com.tinkerforge.*;
4  ...
5  public class Brick_hr4tx implements Runnable {
6   Thread thread;
7   String brick_UID="hr4tx";
8   String HOST_uqB = "192.168.1.2";
9   NFCReader nfc_reader_uqB;
10  int PORT_uqB = 4223;
11  String readerUID_uqB = "uqB";
12  Short currentTagType_uqB = BrickletNFC.
         TAG_TYPE_MIFARE_CLASSIC;
13
14  QuadRelay quadrelay_FZg;
15  String HOST_FZg = "192.168.1.2";
16  int PORT_FZg = 4223;
17  String relayUID_FZg = "FZg";
18     ...
19     DisplayOled display_dDg;
20     Beeper beep_fds;
21     ...
22  public Brick_hr4tx() {
23   nfc_reader_uqB = new NFCReader(...);
24   quadrelay_FZg = new QuadRelay(...);
25   display_dDg = new DisplayOled(...);
26   beep_fds = new Beeper(...);
27  }
28
29  @Override
30  public void run() {
31   try {
32          ...
33    System.out.println(MessageFormat.format("Waiting␣for
         ␣scan", ...);
34    ArrayList<MyBricklet> bricklets=new ArrayList<
         MyBricklet>();
35    bricklets.add(quadrelay_FZg);
36    nfc_reader_uqB.initializeReader(...,bricklets);
37   } ...
38  }
39 ...
40 }
```

Listing 1. *Single Brick Java code*

Since this brick has basically different types of connected bricklets, e.g., NFC Reader, QuadRelay, etc, lines 9-20 declare the variables needed for initializing these modules. For instance, lines 9-12 declare the NFC Reader variables, lines 14-17 the ones for the relay and the following lines 19-20 the variables for the speaker and the display. Then, in the constructor of the Brick, lines 22-27, all the bricklets part of the chosen architecture are initialized using Java classes included in the project that we developed. It is worth noticing that each brick can be configured with different bricklets, depending on the architect and specific needs. To be clear, one brick can be equipped with display and another with a speaker, depending on the noise level of the area. In particular, this will make the initialization code different from a brick to another.

```
1  public class QuadRelay {
2  ...
3   public void OpenRelay(String HOST, int PORT, String
        UID, long millis, boolean[] relays) {
4    IPConnection ipcon = new IPConnection(); // Create IP
          connection
5    BrickletIndustrialQuadRelayV2 iqr = new
         BrickletIndustrialQuadRelayV2(UID, ipcon);
6
7    try {
8     ipcon.connect(HOST, PORT);
9       // Turn relays on/off
10    for(int i=0;i<relays.length;i++){
11     if(relays[i]){
12     iqr.setChannelLEDConfig(i,
           BrickletIndustrialQuadRelayV2.
           CHANNEL_LED_CONFIG_SHOW_HEARTBEAT);
13     Thread.sleep(millis);
14     iqr.setChannelLEDConfig(i,
           BrickletIndustrialQuadRelayV2.
           CHANNEL_LED_CONFIG_OFF);
15    }
16     relays[i]=false;x
17    }
18    ipcon.disconnect();
19      }
20   ...}}
```

Listing 2. *Snippet of the QuadRelay Java Code*

Listing 2 reports a snippet of the method written for the QuadRelay component, to open and close the selected channel. This method basically takes as input the position of the channel we want to open and it blinks for the selected timelapse (line 12). Then, it opens the relay and further closes it. We omit the Java classes initializing the OledDisplay and the Speaker for lack of space.

```
public class NFCReader extends MyBricklet{

 public void initializeReader(...)  {

 IPConnection ipcon = new IPConnection();
 final BrickletNFCRFID nr = new BrickletNFCRFID(
      readerUID, ipcon);

 ipcon.connect(HOST, PORT); // Connect to brickd

 // Add state changed listener
  nr.addStateChangedListener(new BrickletNFCRFID.
       StateChangedListener() {
   public void stateChanged(short state, boolean idle)
        {
    try {
     if(idle) {
      nr.requestTagID(currentTagType);
      Thread.sleep(500);
     }
     if(state == BrickletNFCRFID.
          STATE_REQUEST_TAG_ID_READY) {
      // getting tag id
      BrickletNFCRFID.TagID tagID = nr.getTagID();
      String tag =  Integer.toHexString(tagID.tid[0]);
      for(int i = 1; i < tagID.tidLength; i++) {
       tag +=  Integer.toHexString(tagID.tid[i]);
      }
      try {

      System.out.println("Put␣Your␣logic␣here,␣detected
           ␣tag␣"+tag+ "␣by␣reader:␣"+readerUID);

      for (MyBricklet myBricklet : bricklets) {
```

```
30
31      if(myBricklet.getClass()==QuadRelay.class){
32
33      QuadRelay relay= (QuadRelay)myBricklet;
34
35      relay.OpenRelay();
36       }
37     }
38    } ...
39  });
40  // Start scan loop
41  nr.requestTagID(currentTagType);
42  System.out.println("Press␣key␣to␣exit"); System.in.
        read();
43  ipcon.disconnect();
44 }
45 }
```

Listing 3. *NFCReader class*

Lines 29-38 of listing 1 reports the main task of the brick and its main bricklet, namely the NFC reader (whose code is reported in listing 3). The NFC component is initialized with a state change listener that will be executed when a NFC tag touches the NFC reader (lines 12-39 of listing 3). In this case, line 27 prints a message saying to customize this method with a conditional statement for authorizing or not the tag, and it starts the communication with the other bricklets. Line 35 reports a call to the `OpenRelay` method that will execute the method reported in listing 2, previously explained (the default channel to be opened is specified in the initialization). The rest of the code, which is omitted here, will print on the display a message reporting the tag id of the client and it will trigger the sound emission of the speaker. The state changed listener assures that the NFC tag will remain waiting for the next tag, except for possible exceptions. Lines 27-36 (of listing 3) should be customized in order to support the communication with the access control system, for instance by making a REST call with the tag id to check if it is authorized.

```
package tinkerForgeGen;
...
public class InitializationModule{
 //      Initialize the modules with multiple threads
 public static void main(String args[]) throws
     Exception {
  try {
     Brick_hr4tx brick_hr4tx= new Brick_hr4tx();
     brick_hr4tx.start();
     ...
     Brick_d32d3 brick_d32d3= new Brick_d32d3();
     brick_d32d3.start();
   } ...
 }
}
```

Listing 4. *Main initialization for the Bricks*

Eventually, listing 4 is responsible for starting the threads for each brick in the infrastructure. In our case study we considered two bricks, but there are of course more in real cases. This simple infrastructure require basic knowledge of Java (since it has been chosen as target language) and the code implemented for the presented basic functionalities can be estimated in terms of lines of code (LOC) as in table I. The initialization module strongly depends on the number of bricks involved in the infrastructure, as the first rows depend from the number of bricklets mounted per brick. We created a Java class per brick, since each brick can vary the mounted bricklets, and generic class for the different types of bricklet.

TABLE I
ESTIMATION OF THE LINES OF CODE.

| Java components | LOC |
|---|---|
| Bricks + QuadRelay + DisplayOled + Beeper + NFCReader | 400 |
| InitializationModule | 20 |

Of course, the numbers listed in table I can easily grow depending on the size of the infrastructure, and developers will tend to copy and paste snippets from a component to another. This activity is error-prone and time consuming, and it might be detrimental to the whole infrastructure. However, it can be automated by exploiting model driven techniques. To alleviate this issue, in section III we present a model driven approach to partially automate this process.

## III. CONTRIBUTION

To better explain the design of the chosen infrastructure and to enable the code generation, we engineered the Tinkerforge architectural modules in a metamodel, partially depicted in Fig 2. We omit other modules we did not use in our running example, to make the picture readable. An *Infrastructure* is composed of *Brick*s, to which an application can connect through its *ip_address*. All the building blocks are identified with the *UID*. A brick can be connected to *Bricklet*s, that can be classified in various types, for instance *In/Out* devices, *Sensor*s and *Extension*s. The *NFC Reader* is of type *Sensor* and it has to declare which *TagType*s it reads for the targeted application. The *Speaker* (beeper) can be set with specific beep duration, volume and frequency. The *QuadRelay*, instead, requires the time lapse for opening the channel set to true. Also *NFC tag*s are part of this metamodel with a unique identifier and all the sectors to read and write. The structure of a tag is divided into sectors which are composed of blocks, but since the application is simply reading the UID of the tag we do not read and write the sectors (for this reason is omitted from the metamodel).

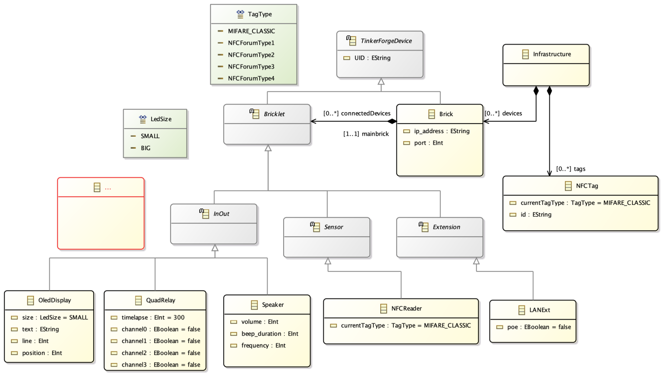


Fig. 2. TinkerForge metamodel

At this point we have a metamodeling layer supporting the definition of two types of models: the infrastructural model (conform to the metamodel in Fig. 2) and the access control system model (conform to the metamodel in Fig. 1). The missing information at this stage is about how the infrastructure is related to the concepts we have in the access control system. For instance, how the room' door is controlled by a specific brick and, in particular, an NFC reader. Moreover, we can also relate NFC tags to the clients in order to keep track of the authorized access. To do that, we conceived a weaving metamodel, namely *Interoperability Weaving metamodel*, that is shown in Fig. 3. A weaving model is a model specification used to capture relationships between model elements [9]. In this metamodel, we basically have two types of links: *InfrastructureLink*, defining how the building blocks of our infrastructure can be related to concepts of the access control system, e.g., room and brick; *AssociationLink* defining that a specific tag has been assigned to a client. This metamodel can be enriched with other types of links and it is still under development. This weaving approach enables a set of automatism that will be discussed in the rest of this section and that we summarize in Fig. 5. Basically, the process starts from the definition of the models of the infrastructure (see Fig. 2) and the access control system (see Fig. 1). These two models can be linked with an *Interoperability Weaving Model* defining the relationships between them.

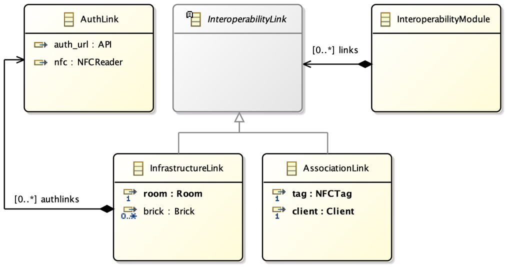
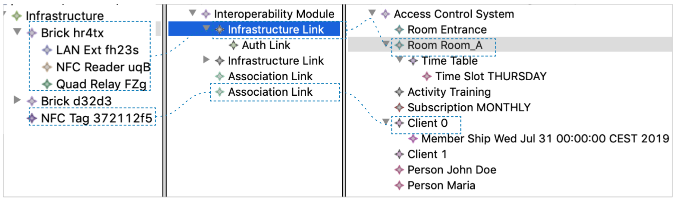


Fig. 3. Interoperability Weaving metamodel


Infrastructure
Brick hr4tx
LAN Ext fh23s
NFC Reader uqB
Quad Relay FZg
Brick d32d3
NFC Tag 372112f5

Interoperability Module
Infrastructure Link
Auth Link
Infrastructure Link
Association Link
Association Link

Access Control System
Room Entrance
Room Room_A
Time Table
Time Slot THURSDAY
Activity Training
Subscription MONTHLY
Client 0
Member Ship Wed Jul 31 00:00:00 CEST 2019
Client 1
Person John Doe
Person Maria

Fig. 4. Interoperability weaving example model

Fig. 4 reports a screenshot of the weaving editor, where an interoperability weaving model has been created to link the exemplary infrastructure with a simple access management model. The dotted lines highlight how the concepts have been linked: for instance, the Brick with UID `hr4tx` has been linked to `Room_A`. Room_A hosts an activity called `Training` on Thursday from 10:00 to 13:00 (visible by exploring the property view of the model). Moreover, we have some associated tags, e.g., the tag with id `372112f5` is associated with client 0, aka John Doe. Not all the properties of the model elements are visible from this screenshot, since we tried to highlight the links defined in the weaving model.

A general process depicting the overview of the approach is reported in Fig. 5. At the Domain Engineering phase, we have the metamodeling layer proposed in previous sections. A code generator has been developed to automatically generate the code interacting with the Tinkerforge APIs as (manually) defined in section II. In fact, an Acceleo code generator has been developed for creating Java code for this application, but it can be easily extended to support other target languages, e.g., PHP. Acceleo[1] is a template-based technology to create custom code generators. This code generator takes as input the *Interoperability Weaving Model*, it navigates to the left and right models to get the right model elements and it generates the code interacting with the Tinkerforge API, exactly as shown in section II. For the sake of clarity, the access control model can be used to generate part of the web application managing the data, as well as the SQL script of the application [10]. This is out of the scope of this paper but we will further extend the approach in the future work.

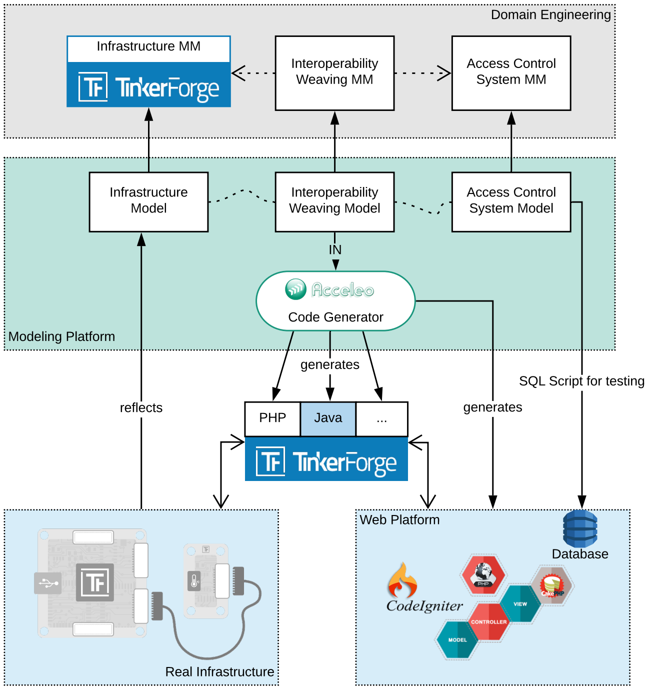


Fig. 5. General overview of the approach.

The code generator is composed of multiple templates. The main template is called *generateElement* (listing 5) and it is responsible for generating the initialization file of the modules.

```
[module generate('http://tinkeforge.org/
      interoperability','http://tinkeforge.org','http://
      cs.ssi.it/access-control')]

[template public generateElement(interoperabilityModule
       : InteroperabilityModule)]
[comment @main]

[ if((interoperabilityModule.links->select(l|l.
      oclIsKindOf(InfrastructureLink)).oclAsType(
      InfrastructureLink).brick.connectedDevices->select(
      e|e.oclIsKindOf(NFCReader )))->size()>0) ]
 [generateNFCReaderCode.../]
```

[1] Acceleo - https://www.eclipse.org

```
8  [/if]
9  [ if((interoperabilityModule.links->select(l|l.
       oclIsKindOf(InfrastructureLink)).oclAsType(
       InfrastructureLink).brick.connectedDevices->select(
       e|e.oclIsKindOf(QuadRelay )))->size()>0) ]
10 [generateQuadRelayCode.../]
11 [/if]
12 ...
13 [generateInitializationCode(interoperabilityModule)/]
14 [/template]
```

Listing 5. *Snippet of Acceleo main template*

Indeed, for each interoperability link of the input weaving, it selects the infrastructure links and for each type of bricklets, connected to the main brick, it generates the corresponding Java class in the workspace. For instance, if an Infrastructure Link is connecting a room to a brick containing an NFCReader and a QuadRelay, it will generate `NFCReader.java` and `QuadRelay.java` (shown respectively in listing 3 and 2). The final call at line 13 in listing 5 is responsible for generating the threads initialization in the main java file invoking the template in listing 6. Here, for each infrastructure link it creates the Brick java class containing the code explained in listing 1 and it generates the code for initializing the bricks contained in the declared infrastructure model, in lines 5-6 in listing 6 (see listing 4 for the produced code).

```
1  public static void main(String args[ '[' /][ ']' /])
       throws Exception {
2   try {
3     [for (link : InfrastructureLink |
          interoperabilityModule.links->select(l|l.
          oclIsKindOf(InfrastructureLink)))]
4      [createBrickComponent(link)/]
5      Brick_[link.brick.UID/] brick_[link.brick.UID/]=
           new Brick_[link.brick.UID/]();
6      brick_[link.brick.UID/].start();
7     [/for]
8    } catch (Exception ex) {
9    ex.printStackTrace();
10   }
11 }
```

Listing 6. *Snippet of Acceleo main template*

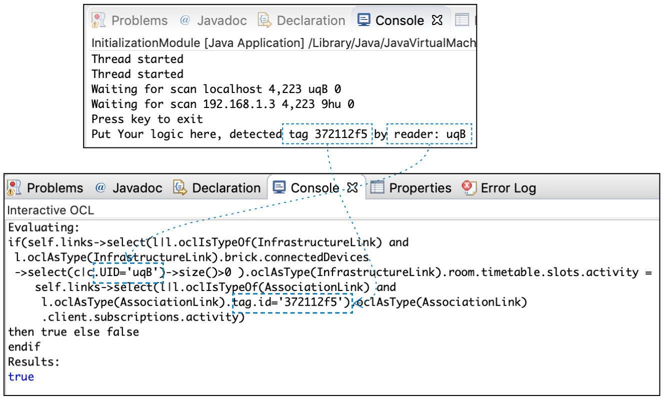
The last template that we report for completeness is `createBrickComponent`, triggered by line 4 of listing 6. Listing 7 reports a snippet of this template, where first of all a selective list of import is generated, depending on the real building blocks declared in the infrastructure. This selectively generates the real package imports needed by the program (see for example lines 1-3). Lines 7-14 is the template part responsible for generating the variables initialization for each type of bricklets connected to the brick and then the constructors of the bricklets are called from lines 17-19 (this code is for QuadRelay, it follows for the other types of Bricklets). Next, it follows the generation of the `run` method, that will be executed by the running thread. In this thread the main task will be to read the NFC tags and if other bricklets are connected using them to orchestrate the infrastructure. The code responsible for the state changed listener, which has been previously shown in listing 3, will be generated in the `NFCReader.java` file that will be called by the `initializeReader` in line 32 of the generator (listing 7).

```
1  [ if(link.oclAsType(InfrastructureLink).brick.
       connectedDevices->select(e|e.oclIsKindOf(QuadRelay
       ))->size()>0) ]
2  import com.tinkerforge.BrickletIndustrialQuadRelayV2;
3  [/if]
4  ...
5  public class Brick_[link.brick.UID/] implements
       Runnable{
6   Thread thread;
7  [for (reader : NFCReader ...))]
8   NFCReader nfc_reader_[reader.UID/];
9   ...
10 [/for]
11 [for (relay : QuadRelay |...))]
12  QuadRelay quadrelay_[relay.UID/];
13     ...
14 [/for]
15  public Brick_[link.brick.UID/](){
16    ...
17    [for (relay : QuadRelay |...))]
18     quadrelay_[relay.UID/]=new QuadRelay(HOST_[relay.
           UID/],PORT_[relay.UID/],relayUID_[relay.UID/],
           millis_[relay.UID/],relays_[relay.UID/]);
19    [/for]
20  }
21  @Override
22  public void run(){
23  try{
24  [for (reader : NFCReader | link.brick.connectedDevices
        ->select(nfc|nfc.oclIsKindOf(NFCReader)))]
25
26       ArrayList<MyBricklet> bricklets=new ArrayList<
             MyBricklet>();
27       [for (relay : QuadRelay |...)]
28       bricklets.add(quadrelay_[relay.UID/]);
29       [/for]
30       ...
31             System.out.println(MessageFormat.format("
                   Waiting␣for␣scan␣{0}␣{1}␣{2}␣{3}",
                   HOST_[reader.UID/], PORT_[reader.UID/],
                    readerUID_[reader.UID/],
                   currentTagType_[reader.UID/]));
32           nfc_reader_[reader.UID/].initializeReader(
                 HOST_[reader.UID/], PORT_[reader.UID/],
                 readerUID_[reader.UID/], currentTagType_[
                 reader.UID/], bricklets);
33      [/for]
34  }...
35 }
```

Listing 7. *Snippet of createBrickComponent template*

The code in lines 27-29 will add to a list of available bricklets the devices connected to the brick, that will be available in the NFCReader code, so that the developer can customize his/her own logic. This specific generated code will be customized with respect to the real infrastructure of connected bricklets, completely specified at the model level.

By executing the generated code, we managed to test the communication mechanism with the infrastructure we modeled. In fact, the screenshot of the log in Fig. 6 (top side) reports the printed detected tag id, e.g., `372112f5`, corresponding to a specific client of the access control system. To verify this, we also executed an OCL query on the access control system model in order to retrieve information about if the tag has been assigned to someone and if this person is actually authorized. The bottom part of Fig. 6, indeed, reports the OCL query executed crossing the detected tag by the NFC reader with UID `uqB` with the clients and activities of the room infrastructured with that brick. In this case, the query returns positive authorization for the client John Doe, which is assigned to that tag in the model. This part has to be integrated in the logic of the application, for instance by including an API call to the authorization url of the access control system. This part has to be manually specified since the possible logics can be multiple, for instance, grant everyone and log the information or check the authorization.



```
InitializationModule [Java Application] /Library/Java/JavaVirtualMach
Thread started
Thread started
Waiting for scan localhost 4,223 uqB 0
Waiting for scan 192.168.1.3 4,223 9hu 0
Press key to exit
Put Your logic here, detected tag 372112f5 by reader: uqB
```

```
Interactive OCL
Evaluating:
if(self.links->select(l|l.oclIsTypeOf(InfrastructureLink) and
 l.oclAsType(InfrastructureLink).brick.connectedDevices
 ->select(c|c.UID='uqB')->size()>0 ).oclAsType(InfrastructureLink).room.timetable.slots.activity =
    self.links->select(l|l.oclIsTypeOf(AssociationLink) and
     l.oclAsType(AssociationLink).tag.id='372112f5').oclAsType(AssociationLink)
     .client.subscriptions.activity)
then true else false
endif
Results:
true
```

Fig. 6. Log file of the execution of the generated code

## A. Discussion

Our approach includes the following limitations.

*Tinkerforge usage.* The presented approach is based on the Tinkerforge platform. Currently, the actual implementation of the specified metamodel does not comprehend all the possible bricklets that Tinkerforge provides. The code generation process has been tested only for the devices we currently have, by making the Tinkerforge platform underused. However, the approach we proposed is general and it can be further extended to support the complete set of bricklets. In particular, to enable the support for a new type of module, it requires two main steps: (i) the development of the code for the new module, which should follow the specification given on the documentation, and (ii) the reverse engineering to placeholders of the code generator, where the code should refer to the specified domain concepts. This process represents a sort of reverse engineering, which the modeler should apply for each target language that has to be supported, in the case of multi-platform code generation.

*Further design models.* The models created with the engineered metamodel allow the specification of *structural characteristics* of the infrastructure. The generated code acts as sample code that the developer might customize to address the logic of the application. For instance, the model does not contain the interaction between the retrieved NFC tags and the authorization system, that can expose an API making authorization data available. The logic of the authorization mechanism, and its conditional statements, might be supported with *behavioural models*, e.g., sequence diagrams, in order to be able to generate the complete application. As future work we plan to extend our approach to embed these models.

*Web application design.* From the *interoperability weaving model* shown in Fig. 3 we could also generate part of the web application that acts as human resources and facility management system, with the related API for the authorization mechanism. This topic has been largely addressed, e.g., in [11], [12], and it can be replicated in our approach, to be able to generate a complete application for access management. We leave as future work the overall application design and realization, since we expect that this does not affect the structural IoT-based infrastructure we realized for this work.

# IV. Related Work

*Access Control Systems* provide a security application to check entries via controlled gates to restricted access areas. Their application is foreseen in both out-door [13], [14] and indoor [15] environments, through diverse technologies. For instance, [13] reports an approach using in-vehicle RFID tags and transponders for roadside to vehicle communications for electronic tolling pricing systems. An automatic toll system using wireless transmission between the vehicles and a fixed station is, instead, documented in [14]. In [15], a distributed access control mechanism, in a smart building scenario is described. An engine embedded into smart objects makes authorization decisions based on user location and access credentials. Differently from our approach, user location data are estimated via magnetometer sensors in the smartphones.

*Model Driven Engineering* (MDE) has been already adopted to deal with the challenges of the IoT domain (e.g., heterogeneity, reusability of software artifacts) [16]. In [17] two different approaches for the development of an IoT application, IoT Mashups and Model-based IoT are presented. The first one, uses existing services/tools to develop applications, while the second one exploits MDE techniques. The work in [17] compares the two approaches and shows that mashup tools are efficient for describing system architectures, message flow and deployment, while model-based approaches have a greater expressiveness to model different points of view and behaviors and to generate code from models for different platforms. In [18] the authors propose an high-level rule-based language to model high-level representations of devices, to manage the interoperability and behaviour of interconnected IoT devices.

*(Semi) Automatic code generation* for architectural models has gained a lot of attention in recent years, even in the IoT domain. In [19], the authors present a Domain-specific Modeling Language for TinyOS, called DSML4TinyOS, whose aim is that of allowing developers to generate architectural code for low power wireless devices. The main purpose of the authors is that of providing a platform-independent modeling framework to abstract from different IoT operating systems, by exploiting model-to-model transformation and code generation.In [20] FRASAD is presented, and it provides a fast way to develop IoT applications by means of model-to-model transformation and code generation, running on top of a node-centric software architecture based on the sensor node domain concept. In [21], the presented approach allows the automatic generation of code running on Arduino components, in the context of situational aware IoT.

*Domain Specific Languages* (DSLs) have been largely adopted to software development in different contexts [22]. In the context of Web applications development, for instance, a DSL for the implementation of dynamic web applications is presented in [23] and further extended in [24]. In the context of access control, we have examples of DSL specifications that try to fill the gap between role-based access control (RBAC) models and policy specification languages. In particular, in [25] the authors proposed a DSL whose semantic can be expressed in terms of a formalization of rule-based access control policies as OCL constraints on the corresponding conceptual model. This approach aims to define policies in a more expressive and clear way and to implement a method that allows the easily reinforcement of these policies. Moreover, in [26], a DSL called SenNet for developing WSN applications is presented. It has been implemented to reduce the complexity of low-level programming details associated with the domain of WSN. [27] represents a further application of DSL in the IoT context to reduce its inherent complexity. This work presents an editor, called DSL-4-IoT, implemented to deal with the high complexity and heterogeneity of WSN and devices. The editor outputs IoT application configuration files which are then executed by a runtime engine (i.e., OpenHAB).

In conclusion, in our approach we profit from the before mentioned benefits of MDE, DSL and code generation based approaches, by additionally combining them with the use of the Tinkerforge platform. This allows the approach to further push and increase modularization, abstraction and code generation features. Moreover, the provided infrastructure is open and extensible to different technologies and device types, besides RFID/NFC.

## V. Conclusions and Future Work

In this paper we presented a Model Driven approach supporting the (semi) automatic code generation for enabling the communication between an IoT infrastructure and an existing web platform for Facility Access Management. This work has been triggered by the need for migrating an old chip card based access control system to a new way of authorizing people with NFC tags. In particular, the approach we provide enables the generation of software components in the selected target language (e.g., Java), by also offering snippets as exemplary code usage, when possible. The objectives of the presented approach are manifold: (i) relieving developers from the inherent complexities of heterogeneous IoT devices, protocols, networks, also in view of a multi-platform evolution of the approach; (ii) making the whole development process less error-prone and time consuming, while exploiting code generation. As future work, we plan to address the limitations discussed in section III-A and to make the generative approach multi-platform, by targeting further languages besides Java, through a wider exploitation of the Tinkerforge platform.

## References

[1] D. C. Schmidt, "Guest NOOPeditor's Introduction: Model-Driven Engineering," *Computer*, vol. 39, no. 2, pp. 25–31, 2006.

[2] D. Di Ruscio, M. Franzago, I. Malavolta, and H. Muccini, "Envisioning the future of collaborative model-driven software engineering," in *International Conference on Software Engineering Companion (ICSE-C)*, 2017, pp. 219–221.

[3] B. Morin, N. Harrand, and F. Fleurey, "Model-based software engineering to tame the iot jungle," *IEEE Software*, vol. 34, no. 1, pp. 30–36, 2017.

[4] L. Gaouar, A. Benamar, and F. T. Bendimerad, "Model driven approaches to cross platform mobile development," in *Proceedings of the International Conference on Intelligent Information Processing, Security and Advanced Communication*, ser. IPAC. ACM, 2015, pp. 1–5.

[5] S. Rehman, R. M. K. Ullah, S. Tanvir, and F. Azam, "Development of user interface for multi-platform applications using the model driven software engineering techniques," in *IEMCON*, 2018, pp. 1152–1158.

[6] V. Coskun, B. Ozdenizci, and K. Ok, "A survey on near field communication (nfc) technology," *Wireless Personal Communications*, vol. 71, no. 3, pp. 2259–2294, 2013.

[7] J. Bravo, R. Hervas, G. Chavira, S. W. Nava, and V. Villarreal, "From implicit to touching interaction: Rfid and nfc approaches," in *Conference on Human System Interactions*, 2008, pp. 743–748.

[8] Tinkerforge, "Website," https://www.tinkerforge.com, 2011.

[9] M. D. Del Fabro, J. Bézivin, and P. Valduriez, "Weaving models with the Eclipse AMW plugin," in *Eclipse Modeling Symposium, Eclipse Summit Europe*, 2006.

[10] A. Cicchetti, D. Di Ruscio, L. Iovino, and A. Pierantonio, "Managing the evolution of data-intensive web applications by model-driven techniques," *Software & Systems Modeling*, vol. 12, no. 1, pp. 53–83, 2013.

[11] A. Cicchetti, D. Di Ruscio, and A. Pierantonio, "Managing Dependent Changes in Coupled Evolution," in *ICMT*, ser. LNCS. Springer, 2009, vol. 5563, pp. 35–51.

[12] M. Brambilla, S. Comai, P. Fraternali, and M. Matera, "Designing web applications with webml and webratio," in *Web Engineering: Modelling and Implementing Web Applications*. Springer, 2008, pp. 221–261.

[13] P. Blythe, "Rfid for road tolling, road-use pricing and vehicle access control," in *IEE colloquium on RFID technology*. IET, 1999, pp. 1–8.

[14] D. S. Breed and V. Sokurenko, "Tolling system and method using telecommunications," 2017, uS Patent 9,691,188.

[15] M. V. Moreno, J. L. Hernández, and A. F. Skarmeta, "A new location-aware authorization mechanism for indoor environments," in *international conference on advanced information networking and applications workshops*. IEEE, 2014, pp. 791–796.

[16] S. Wolny, A. Mazak, and B. Wally, "An initial mapping study on mde4iot," *MDE4IoT - 2nd International Workshop on Model-Driven Engineering for the Internet-of-Things*, pp. 524–529, 2018.

[17] C. Prehofer and L. Chiarabini, "From internet of things mashups to model-based development," in *Computer Software and Applications Conference, COMPSAC Workshops*, 2015, pp. 499–504.

[18] M. Amrani, F. Gilson, A. Debieche, and V. Englebert, "Towards user-centric dsls to manage iot systems," in *International Conference on Model-Driven Engineering and Software Development, MODELSWARD*, 2017, pp. 569–576.

[19] H. M. Marah, R. Eslampanah, and M. Challenger, "Dsml4tinyos: Code generation for wireless devices," in *Proceedings of MODELS 2018 Workshops*, 2018, pp. 509–514.

[20] X. T. Nguyen, H. T. Tran, H. Baraki, and K. Geihs, "Frasad: A framework for model-driven iot application development," in *IEEE World Forum on Internet of Things (WF-IoT)*, 2015, pp. 387–392.

[21] M. Sharaf, M. Abughazala, and H. Muccini, "Arduino realization of caps iot architecture descriptions," in *European Conference on Software Architecture: Companion Proceedings*, ser. ECSA, 2018, pp. 1–4.

[22] B. Langlois, C.-E. Jitia, and E. Jouenne, "Dsl classification," 2007.

[23] D. Groenewegen, Z. Hemel, L. Kats, and E. Visser, "Webdsl : A domain-specific language for dynamic web applications," 2008, pp. 779–780.

[24] D. M. Groenewegen and E. Visser, "Integration of data validation and user interface concerns in a dsl for web applications," *Software & Systems Modeling*, vol. 12, no. 1, pp. 35–52, 2013.

[25] A. Ben Fadhel, D. Bianculli, and L. Briand, "Gemrbac-dsl: A high-level specification language for role-based access control policies," in *Symposium on Access Control Models and Technologies*, ser. SACMAT. ACM, 2016, pp. 179–190.

[26] A. J. Salman and A. Al-Yasiri, "Developing domain-specific language for wireless sensor network application development," in *International Conference for Internet Technology and Secured Transactions (ICITST)*, 2016, pp. 301–308.

[27] A. Salihbegovic, T. Eterovic, E. Kaljic, and S. Ribic, "Design of a domain specific language and ide for internet of things applications," in *International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, 2015, pp. 996–1001.