Exploiting Multi-Level Modelling for Designing and Deploying Gameful Systems

Antonio Bucchiarone
Fondazione Bruno Kessler
Trento, Italy
bucchiarone@fbk.eu

Antonio Cicchetti
IDT Department
Mälardalen University
Västerås, Sweden
antonio.cicchetti@mdh.se

Annapaola Marconi
Fondazione Bruno Kessler
Trento, Italy
marconi@fbk.eu

***Abstract*—Gamification is increasingly used to build solutions for driving the behaviour of target users' populations. Gameful systems are typically exploited to keep users' involvement in certain activities and/or to modify an initial behaviour through game-like elements, such as awarding points, submitting challenges and/or fostering competition and cooperation with other players. Gamification mechanisms are well-defined and composed of different ingredients that have to be correctly amalgamated together; among these we find single/multi-player challenges targeted to reach a certain goal and providing an adequate award for compensation. Since the current approaches are largely based on hand-coding/tuning, when the game grows in its complexity, keeping track of all the mechanisms and maintaining the implementation can become error-prone and tedious activities. In this paper, we describe a multi-level modelling approach for the definition of gamification mechanisms, from their design to their deployment and runtime adaptation. The approach is implemented by means of JetBrains MPS, a text-based meta-modelling framework, and validated using two gameful systems in the Education and Mobility domains.**

***Index Terms*—Multi-Level Modelling, Model-Driven Engineering, MPS, Gamification Engine**

I. Introduction

Studies on human behaviour generally agree on the effectiveness of gamification as a self-motivation tool. In particular, it is widely accepted that, in order to promote behavioural changes typically hard to achieve through enforcement rules, building up a game-like scenario in which such changes are adequately rewarded can be successful. That is the reason why there is a growing exploitation of gameful systems in disparate contexts [1], including promotion of healthy habits, self-motivation for education, encouragement to use more public transportation, to recycle, to reduce food waste, etc. Gamification mechanisms are quite standardised in their construction elements: in general there is one or more challenges for which the player gets a reward when the challenge is accomplished. However, a very important condition for the game to be effective is to keep engaged its players: a too easy/difficult challenge might quickly lose interest and hence make the game goals to fail [2]. In this respect, setting up all the aspects of a game can become tedious and error-prone, especially when dealing with a lot of users and the game combining multiple, interacting challenges.

Gameful systems development requires therefore a well-founded software engineering approach [3], [4]. Current approaches tackle the design and development at a very low level of abstraction. In some cases it is possible to define the main components, as challenges, rewards, players, etc. in a more user-friendly way, however the core gaming behaviour has to be coded in an appropriate programming language (usually rule/event-based). As a consequence, there exists an abstraction gap between domain experts (both for gaming and for the field of exploitation of the game), and the way in which the game is developed. Practically, domain experts define the game requirements and its main components, while programmers translate the desired behaviours into corresponding code routines. This gap can cause unexpected game deviations to be detected late, if ever discovered. Moreover, it makes very difficult to maintain and evolve a running game.

With the aim of raising the level of abstraction of gamification mechanisms, we propose a well-defined set of languages for designing a game, its main components, and the behavioural details. In this respect, we introduced a multi-level way of development, given the inherent characteristic of games definition: at the highest level of abstraction a game is defined by its main components, keeping in mind a certain gamification strategy; at the next (lower) level, each of the components takes form as a set of game modules; starting from such a library of modules, each component can be instantiated into a concrete game component; as lowest level of abstraction, we consider the game at runtime. This chain of instantiation levels is better expressed and supported by means of multi-level modelling approaches rather than the traditional two-level modelling [5].

The implementation of the gamification design and deployment is realised by means of JetBrains MPS (briefly, MPS)[1]. MPS is a meta-programming framework that can be exploited as modelling languages workbench, it is text-based, and provides projectional editors. The choice of MPS is due to the inherent characteristics of game definition languages, which are by nature collections of rules and constraints. In this respect, graphical languages do not scale with the complexity of the rules. Moreover, MPS smoothly supports languages

[1] http://www.jetbrains.com/mps/

embedding, such that our multi-level definition of gamification solutions is easily implemented.

As a validation of our approach, we re-designed and deployed two real gamification applications in the education and mobility domains. Our experience shows that by raising the level of abstraction:

- the complexity of the design of gamification applications is reduced;
- the quality of the deployed applications is enhanced;
- runtime adaptation is enabled.

A. Structure of the Paper

The remainder of the paper is organized as follows: Section II presents the basics about gamification and the metamodelling framework used in the proposed approach. Starting from this preliminary information, Section III illustrates the motivations and summarizes the contribution of the work. Section IV presents the Gamification Design Framework (GDF) with all its languages, while Section V shows the applications used to validate the framework and its prototype implementation. In Section VI lessons learned and future investigations are discussed. Section VII surveys related works and Section VIII concludes the paper.

II. Background

A. Gamification

The term gamification has been introduced in the early 2000s [6] and has as central idea the usage of game elements in non-entertainment application domains to foster motivation [7]–[9]. There is a considerable amount of literature concerning gamification concepts [10], [11], related taxonomies [12], [13], and literature reviews [1]. Analysing this literature, we can characterize a *gameful system* as a software artifact that embeds a *gamification process* composed by the following components: *game elements*, *game mechanics* and *game dynamics*. As depicted in Figure 1, these components form a layered structure starting from the *game elements* that define the game concepts used in the application, passing through the *game mechanics*, that are used to evolve the players and application status, and ending up with the *game dynamics* devoted to the ultimate goals of introducing the gamification into a software system. In the following we present details of the three components and their relations, giving some concrete examples.

B. Game Elements

Game Elements are the basic building blocks of gamified applications [10], [14] and are defined to specify how the players should interact with the application to reach the ultimate goals. *Points* are basic elements of gamified applications and allow tracking and rewarding players for the successful accomplishment of specific activities using the application (e.g., number of tracked journeys using a shared bike). Each player, accumulating a specific amount of points or executing a particular *action*, makes progress in the general ranking. *Badge* and *Badge Collection* elements are awarded to the player after

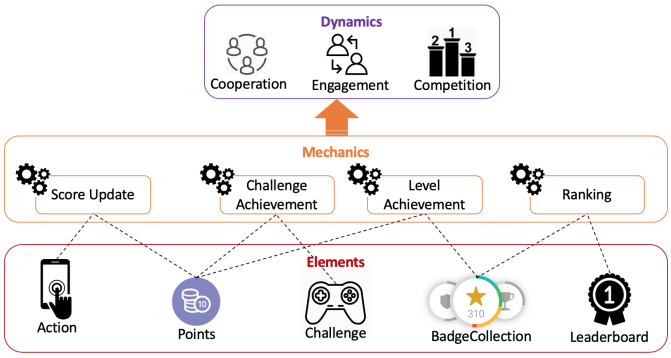

Fig. 1. Gamification Concepts - Elements, Mechanics and Dynamics.

achieving a certain level (e.g., a set of sustainable trips in a limited time) or after finishing a specific task (e.g., invite a friend). Badges symbolize the players merits with a visual status symbol (e.g., green leaves) and indicate how the players are performing. *Leaderboards* rank players or teams and help to determine who performs best in a certain period of the application execution (i.e., weekly, monthly competition). This element is used to create competition among players and to increase their level of engagement and can have a positive effect on participation.

Challenges request of attaining a certain goal hindered by one or more constraints and that results in an extraordinary reward (e.g., at work without car or a certain number of kilometers using a shared bike). The reward for each challenge is calculated according to the estimated difficulty of the challenge itself, which considers the effort that the players are supposed to make to win it on the base of their skills.

In addition to the previous ones, there are many other elements available, such as *levels*, *paths*, *stories*, *feedback*, *progress*, etc.

C. Game Mechanics

Game Mechanics are the set of *rules* that specify how the game should evolve for its participants (e.g., students, citizens or employees). Through these rules the game designer specifies when and how the state of the game and therefore of the players evolves. This evolution happens every time that a game *action* is executed (e.g., bike trip tracking) and has as effect the update of a set of connected game *points* (e.g., shared bike trips, kilometres by bike, etc.). This *mechanics* type is called *Score Update* and defines the semantics of a player's points update. *Challenge Achievement* mechanic is used to define "when" a specific challenge has been achieved (e.g., "do at least 5km by bike") and "how" the player status should evolve (e.g., "to win a bonus of 120 Green Leaves points"). *Level Achievement* mechanic has as main goal to provide players and teams with a level in the game that is proportional to the amount of points they gathered, and represents their level of expertise in the game. For example it can represent how experienced in being green citizens the players are (e.g., Green Soldier, Green Warrior, Green Guru).

Finally, the *Ranking* mechanic is used to understand who performs best in a certain activity and is an indicator of social comparison that relates the single player/team performance to the performance of other players/teams.

D. Game Dynamics

Game Dynamics represent the high-level aspects of a game that have to be considered and managed, but not directly implemented in it. A game dynamic can also be defined as the "emergent" behavior that arises using a gameful system, when the mechanics are used. Each game can provide a single dynamic or a combination of them and they are used to learn the game benefits or its drawbacks and possibly adapt the mechanics and fine-tune the game's overall goal.

Example of game dynamics are:

- *Engagement*: in gamified systems, particular attention is payed to keep users in a state of flow [2], achieved by customized challenges. This dynamic will happen any time that a challenge is defined, assigned and accepted by specific players. The goal behind it is to improve or maintain the players performance without frustrating them with unattainable goals.
- *Competition*: it emerges, for example, every time that players compete against each other to be first in the final *Ranking*. In this case the *Ranking* mechanic is specified in a such a way that the winner is awarded with a prize as for example: theatre or sport tickets and discounts.
- *Cooperation*: this dynamic emerges e.g. when players decide to work together to overcome a challenge. Players are given a common goal and, not matter the individual contribution, if they reach the goal before a multi-player challenge expires they all win. The purpose of this dynamic is to foster cooperation between players.

E. Gamification Engine

A *Gamification Engine* is a software framework responsible for the execution of the game associated with the gameful application. It is usually a *rule* execution system[2] able to execute a rule set, which constitutes the implementation of the game logic. This logic is specified by a *gamification designer* and mainly includes the game mechanics needed for the specific game. In particular, each mechanic is represented by a set of rules that are fired in response to incoming instances of the defined *gamifiable actions*. These are game-relevant actions and events that occur as the player interacts with the application (i.e., mobile app). The game state evolves executing the rules and manipulating the value of the associated gamification elements (i.e., points, challenges, etc.). For example let's consider a rule assigning a *badge* to a player: since it does not make sense for a player to collect the same badge repeatedly, the rule implements a guard to check whether the player already has that badge in her collection. Besides changing the state of the game, the effect of a rule execution can also produce additional notifications as for example the presentation of the updated game state to the player.

[2]The gamification engine used in this paper uses the open source DROOLS (http://www.drools.org) rule engine.

F. Jetbrains MPS: a text-based metamodelling framework

Meta Programming System (MPS) by JetBrains is a text-based meta-programming system that enables language oriented programming [15]. MPS is open source and is used to implement interesting languages with different notations [16]. In particular, based on MPS *BaseLanguage* it is possible to define new custom languages through extension and composition of *concepts* [17]. A new language is composed by different *aspects* making its specification modular and therefore easy to maintain [18], [19]. Notably, the *Structure Definition* aspect is used to define the Abstract Syntax Tree (AST) of a language as a collection of *concept*s. Each concept is composed of properties, children, and relationships, and can possibly extend other concepts. The *Editor Definition* aspect deals with the definition of the concrete syntax for a DSL: it specifies both the notation (i.e., tabular, diagram, tree, etc.) and the interaction behavior of the editor. The *Generators Definition* aspect is used to define the denotational semantics for the language concepts. In particular, two kinds of transformation are supported: (1) AST to text (model-to-text), and (2) AST to AST (model-to-model). Other aspects like the *Type System Definition*, the *Constraints Definition*, etc. are provided. For the sake of brevity, we refer the reader to [20] and [21].

III. Motivation and contribution

As clarified so far, each gamification solution is the result of a combination of well-defined ingredients that should be carefully combined to gain and keep the motivation of the intended target players. This objective can be challenging when dealing with applications involving large number of users, and combining a variety of behavioural change goals. In this respect, designing and deploying gameful systems needs adequate software engineering practices [4], which however are largely missing: as we will discuss more extensively in Section VII, most of the existing solutions either provide ready-made games with scarce customization chances, or they demand low level of abstraction tuning comparable to writing code in a programming language. As a result, current approaches introduce an abstraction gap between domain experts and concrete solutions, which has to be closed by programmers. Moreover, programmers themselves typically need to "translate" these applications in terms of rules for rule-/event-based mechanisms. The consequences are: (a) increased error-proneness of the implementation with the growth of the game complexity and its needs of maintenance and evolution; (b) difficulty in monitoring the evolution of the game and possibly applying runtime adjustments; (c) decreased chances of reusing game elements in other scenarios. Notably, a developer might erroneously set a wrong update for a certain score, making a challenge too easy/complicated or even impossible to achieve; or the game designers would realise that in certain

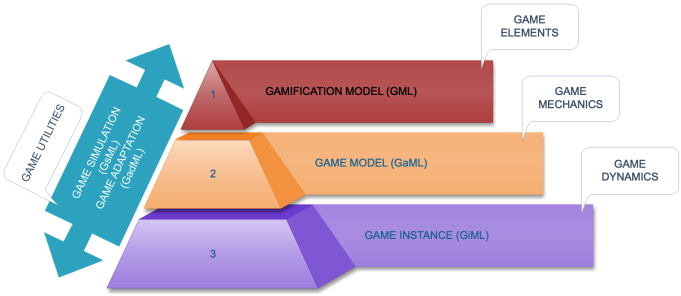

Fig. 2. The Gamification Design Framework (GDF).

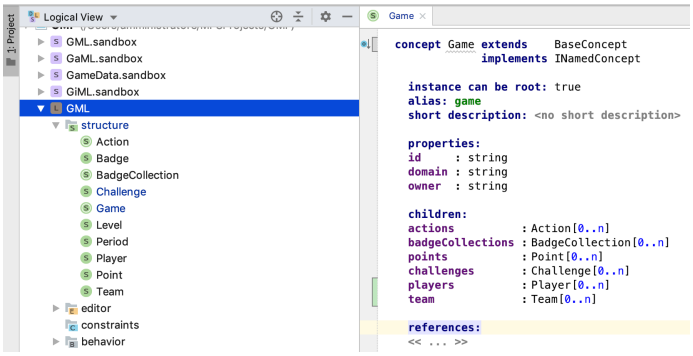

Fig. 3. The Gamification Model Language (GML).

contexts the players progress through the game levels at a too slow pace.

We aimed at tackling the complexity of the mentioned and other problematic cases by raising the level of abstraction in the design and deployment of serious games. This paper reports our experiences in designing and realizing a MDE approach for gamification. In particular, we propose to exploit: separation of concerns [22], where each concern is one of the gamification components shown in Figure 1; domain-specific languages (DSLs) for each concern; automated generation of implementation code. Moreover, our solution is based on a multi-level modelling [5] approach such that: a gamification model defines the elements by which a certain game model will be designed; in turn, the game model specifies the concepts through which a game instance will be described; based on the game instance, the code for deploying the game is automatically generated. These mechanisms are realised by means of MPS — a text-based language workbench — and validated against two real life gamification applications.

IV. Gamification Design through Multi-Level Modelling

In order to tackle the complexity of gamification design we propose a Gamification Design Framework (GDF). The framework provides a modular approach that can be customised for different gameful systems and reflects the *gamification process*, introduced in Section II-A. In particular, it provides a modelling language for each component (i.e., Game Elements, Mechanics and Dynamics) plus *game utility* languages devoted to the management of *simulation* and runtime *adaptation* phases of a gameful application. The GDF architecture has been specified and is composed by different *modelling layers*, each of which defined on the basis of the layer above; moreover, *utility layer*s are orthogonal to the others in the sense that they can be defined on the basis of any of them. A graphical representation of this stack is shown in Figure 2. It is worth noting that the top layer (GML in the picture) is defined on the basis of the MPS language workbench, while code generation is performed by means of the last two layers (GaML and GiML in the picture, respectively). Given the architecture of GDF and the relationships between the gamification components, exploiting a multi-level modelling approach adequately fits our purposes. In fact, the framework starts from very general gamification concepts, which could be part of any application, and it gets progressively more specific through the instantiation of the generic concepts into concrete game elements, included only in a particular game and a corresponding implementation. Notably, generating code for a different target gamification engine would require the sole modification of the generators (see later in this section for more details on the code generation), while all the remaining part of the GDF components would stay the same. The same kind of relationships could have been defined by means of traditional two-level (meta-)modeling approaches, however it has been extensively discussed in existing literature that such type/instance facet requires workarounds to be successfully implemented [5], [23]. For example, GaML is built-up and used by instantiating elements defined in GML, hence in this respect a game definition is correct by construction from a type perspective. The same cannot be guaranteed by a two-level modeling approach, which would require additional checks and constraints to ensure that a type is always related with a correct instance of itself. These issues are amplified whenever new types would be introduced (e.g., new game elements in our case), since the correctness checks shall be manually added to the language definition.

The following sections provide detailed descriptions of each of the layers in Figure 2 with the aim of clarifying how GDF allows to partition the problem of gamification design in simpler sub-problems which are later on re-amalgamated to generate the target application code.

A. The Gamification Model Language (GML)

Gamification is generally referred to as the use of elements characteristic of games in *non-game* contexts [10]. Therefore, the top layer of our solution is represented by a core language defined to introduce the essential elements to describe a gameful system. Figure 3 shows an excerpt of the concepts defining the abstract syntax of the Gamification Modelling Language (GML)[3]. In particular, a `Game` concept comprises a set of properties (i.e., `id`, `domain`, and `owner`) that characterise the specific gameful system, and a set of *children* representing its core ingredients. As described in Section II-A,

[3]The complete specification of the languages together with their graphical rendering are included in the GitHub repository of the framework, available at https://github.com/antbucc/GDF

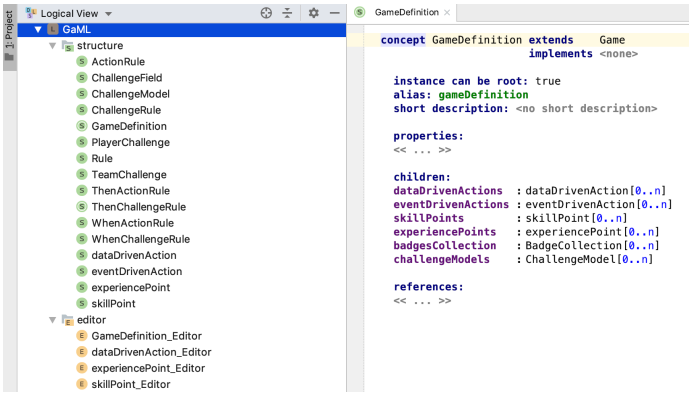

Fig. 4. GameDefinition Concept of the GaML.

each gamified application must be designed defining a set of *game elements* devoted to specify how the players should interact with the application. The `children` part of the `Game` concept of GML has been defined for this purpose. This core language provides the basic building blocks on top of which all the subsequent modelling layers can be described. As already mentioned, GML is itself an instance of a modelling language, that is the MPS base language.

B. The Game Model Language (GaML)

Based on the core ingredients in GML, the Game Model Language (GaML) relies on *Game Mechanics* and allows the game designer to design a certain concrete game. An excerpt on the concepts defined for GaML are depicted in Figure 4. At this level of abstraction the designer can specify in more details how the game components are assembled to create a gamification application. In particular `dataDrivenAction` are operators that act on data (i.e., kms, legs, etc.) while `eventDrivenAction` are related to specific events (i.e., surveys, check-in, etc.). `skillPoint` are points used to denote a users ability in a specific area, while `experiencePoint` are points used to quantify a players progression through the game. Finally, `badgeCollection` and `challenge` represent game elements that each players can collect and achieve, respectively.

It is very important to notice that in this way GaML allows the designer to specify game components which are reusable in different gamification scenarios. Notably, the abstract concept `experiencePoint` can take the concrete forms of `pedibus_distance` or `Walk_Km`, depending on the target application.

GaML also enables the definition of a new game and its deployment in a target gamification engine. This part is performed automatically by means of an appropriate generator (see Figure 5 for an excerpt). In particular, it is a part of the language specification that defines the *operational semantics* of the `GameDefinition` concept in the GaML language. As illustrated in Figure 5, generators specifications are given by means of template mechanisms. Templates are written by using the output language (i.e., Java in our case), and are parametric with respect to the elements retrievable from the input model

```
root template
input GameDefinition

public class $[map_game] {

  public static void run() {
    try {
      GameDTO game = new GameDTO();

      // add Game Info
      game.setName("$[gameName]");
      game.setId("$[gameId]");
      game.setOwner("$[gameOwer]");

      List<string> actions = new ArrayList<string>();

      //  Add Game DataDriven Actions
      $LOOP$[actions.add("$[actionName]"); ]

      //  Add Game EventDriven Actions
      $LOOP$[actions.add("$[actionName]"); ]

      //  Add Game Skill Points
      List<PointConcept> points = new ArrayList<PointConcept>();
      long DAY = 60 * 60 * 24 * 1000;
      long WEEK = DAY * 7;
      Date start = new Date();

      $LOOP$[{
        PointConcept p = new PointConcept("$[pointName]");
        var periodType = "$[periodType]";
        if (periodType == "weekly") {
          p.addPeriod(periodType, start, WEEK);

        } else if (periodType == "daily") {
          p.addPeriod(periodType, start, DAY);
        }
        points.add(p);
      }
```

Fig. 5. GaML Generator.

through *Macros*, denoted by the `$` symbol. Notably, in the template shown in Figure 5 the `$LOOP$` macro is used to iterate over the sets of various game elements defined in the game definition (i.e., dataDrivenActions, eventDrivenActions, skillPoints, etc.) and are used to generate the corresponding Java code. The outputs of GaML generator are Java classes that completely define a game, ready to be deployed in the gamification engine.

C. The Game Instance Model Language (GiML)

A gamification engine is responsible for the execution of one or more instances of multiple games that may run concurrently. Therefore, we introduced GiML, a language that relies on the *Game Dynamics* and is used to specify the instantiation of the different games originating from the same `GameDefinition` as defined in GaML. Game instances differ from one another by the set of *Teams* and *Players* that play a game. These concepts have been defined as part of `Environment` concept, children of the `GameInstance` concept, as depicted in Figure 6. The `Environment` is used to define the set of *teams* and corresponding *players* that play a certain instance of a *game definition*.

Similarly to GaML, a *generator* takes as input GiML models (i.e., the `GameInstance` concept as depicted in Figure 7) and derives Java classes used to create corresponding game instances. In this case the template is composed by two main methods: *initEnv* and *defineGame*. The former is used to generate the code needed to instantiate the set of teams and players that will take part in the game (e.g., specific classes and students of a school); the latter is used to instantiate the

```
concept GameInstance extends    BaseConcept
                     implements INamedConcept

  instance can be root: true
  alias: gameInstance
  short description: <no short description>

  properties:
  gameInstanceId : string

  children:
  environment : Environment[1]

  references:
  gameDefinition : GameDefinition[1]
```

Fig. 6. GameInstance Concept of the GiML.

```
root template
input GameInstance

public class map_gameInstance extends GameTest {

  private PlayerService playerSrv;
  private static final String GAME_ID = "$[gameID]";


  @Override
  public void initEnv() {
    //  add all the Teams defined for this game instance

    $LOOP$[{
      TeamState team = new TeamState(GAME_ID, "$[teamName]");
      List<string> members = new ArrayList<String>();
      $LOOP$[members.add("$[playerName]"); ]
      playerSrv.add(team);
    }]

  }
  @Override
  public void defineGame() {
    //  add all the GameDefinition elements
    List<String> actions = new ArrayList<String>();

    //  Add Game DataDriven Actions
    $LOOP$[actions.add("$[actionName]"); ]
    // ......
```

Fig. 7. GiML Generator.

set of game elements from a specific `gameDefinition` (i.e., actions, points, badge, challenges, etc.). This is done iterating on the various sets defined in the `environment` and `gameDefinition` models through the `$LOOP$` macros.

D. The Game Simulation Language (GsML)

When a game instance is running, the game state changes whenever one of the mechanics defined in the game (i.e., score update, challenge achievement, etc.) is used. In particular, this means that one of the game rules defined using GaML is executed in a specific instance defined through GiML. Therefore, the game state evolves as the righthand side of the game rules prescribe to manipulate the object base through the gamification engine services. Based on this, our approach provides support for simulating the behaviour of a running game under certain conditions, by means of the GsML component. It is part of the *Game Utility* component of the GDF (see Figure 2) and its core concept is represented by the `GameSimulation` element depicted in Figure 8. This concept is composed by a `GameDefinition` and a set of `SingleGameExecution`. In turn, each game execution is made-up of a `Team` and/or a `Player` that can execute an `actionInstance` or a `challengeInstance` (see Figure 9). In this way, GsML allows to model specific game scenarios together with triggered mechanisms, and hence to simulate specific game state changes (e.g. for testing purposes).

As for the previous components, also GsML is equipped with a generator to map the `GameSimulation` in a piece of code that can be executed by the gamification engine.

The implemented generator is depicted in Figure 10: the *defineExecData* method generates the set of game executions starting from the defined *GameSimulation*. Each simulation is specified by the target action type (i.e., action, challenge, etc.) executed by a specific player, with well defined values assigned as action data (i.e., the *payload* variable in Figure 10). To assign the specific action to players, we have also defined a property macro able to randomly select a player for each game execution.

E. The Game Adaptation Language (GadML)

The gamification engine exploited by the GDF includes a *Recommendation System (RS)* able to generate challenges tailored to each player's history, preferences and skills. In this context, the GadML language is used whenever a new game content (i.e., a new challenge generated by the RS), has to be assigned to a specific player on-the-fly. The core concept of this language is the `GameAdaptation` element. While the `gameId` and `playerId` are general parameters of a game adaptation, Figure 11 shows the `newChallenge` concept that extends a simple game adaptation defining the `ChallengeModel`, the `ChallengeData` (i.e., *bonusScore*, *virtualPrice*, etc.) and `ChallengeDate` (i.e., start and termination date of the new challenge).

As for the previous components, also GadML is provided with a generator to map the `GameAdaptation` in a piece of code that, if executed, injects the game adaptation (i.e., the new challenge) in the specific player game instance. In this respect, GDF provides a third-party adaptation [24], since the generated applications are currently not able to adapt themselves to different players' behaviors and/or game evolution patterns.

The implemented generator is depicted in Figure 12: it retrieves all the elements of a `newChallenge` definition, i.e. *gameId*, *challengeName*, *ChallengeData* and the *playerId*, and by invoking the `saveGameUsingPOST` method it assigns the new challenge to a specific player.

V. Case Studies and Implementation Details

As a first iteration towards realizing the framework proposed in this paper, we developed a prototype implementation of the GDF. This prototype includes all the functionalities described in Section IV and has been experimented using two Smart City gamified applications in the Education and Mobility domains. In the following sections, we first introduce the two applications used to validate our proposal and then we present the technical details of GDF implementations.

```
concept GameSimulation extends    BaseConcept
                       implements INamedConcept

  instance can be root: false
  alias: <no alias>
  short description: <no short description>

  properties:
  << ... >>

  children:
  listOfExecutions : SingleGameExecution[1..n]

  references:
  game : GameDefinition[1]
```

Fig. 8. GameSimulation Concept of the GsML.

```
concept SingleGameExecution extends    BaseConcept
                            implements <none>

  instance can be root: false
  alias: singleGameExecution
  short description: <no short description>

  properties:
  << ... >>

  children:
  << ... >>

  references:
  team              : Team[1]
  player            : Player[0..1]
  actionInstance    : ActionInstance[0..1]
  challengeInstance : ChallengeInstance[0..1]
```

Fig. 9. Single Execution of a Game Simulation.

```
root template
input GameSimulation

public class map_GameSimulation extends GameTest {
  private static final String GAME_ID = "${gameID}";
  @Override
  public void defineExecData(List<GameTest.ExecData> execList) {

    $LOOP$ {
            Map<String, Object> payload = payload = new HashMap<String, Object>();
            $LOOP$ payload.add("${dataType}", "${dataValue}"); }
            GameTest.ExecData input = new GameTest.ExecData(GAME_ID, "${actionType}", payload);
            // select a player randomly to assign a specific simulation action
            input.setPlayerId("${playerId}");
            execList.add(input);
    }
  }
```

Fig. 10. GsML Generator.

```
concept newChallenge extends    GameAdaptation
                     implements <none>

  instance can be root: true
  alias: newChallenge
  short description: <no short description>

  properties:
  << ... >>

  children:
  challengeModel : ChallengeModel[1]
  challengeData  : ChallengeData[1]
  challengeDate  : ChallendeDate[1]

  references:
  << ... >>
```

Fig. 11. newChallenge Concept of the GadML.

```
root template
input newChallenge

public class newChallenge_map {
  public static void run() throws ApiException {

    ChallengeModelControllerApi challengeApi = new ChallengeModelControllerApi();
    ChallengeModel newChallenge = new ChallengeModel();
    newChallenge.setGameId("${gameId}");
    newChallenge.setName("${challengeName}");
    List<String> challengeData = new ArrayList<String>();

    $LOOP$ {
            challengeData.add("${challengeData}");
           }
    newChallenge.setVariables(challengeData);
    challengeApi.saveGameUsingPOST(newChallenge, "${playerId}");
  }
}
```

Fig. 12. GadML Generator for a new player challenge.

A. The Kids-Go-Green Application

Kids-go-Green (KGG) [25][4] is an application designed for primary school classrooms (age group 6-11) for supporting active and sustainable mobility habits, both at personal and collective level. The core part of KGG is a virtual journey undertaken by students using the sustainable kilometres travelled in real life in their home-school commuting. Every day each class accesses the KGG Web application and fills the *Mobility Journal*. The children indicate the way they reached school that day by expressing a specific *sustainable transportation mode* (i.e., by foot, bike, walking bus, or school bus). The number of kilometres travelled each day by the children in their trips to school contributes to the progress on a *virtual journey*. The latter is created by the *teachers* and includes a final destination and a set of intermediate stops that are locations in the real-world. The journey can be defined according to the interests and the educational needs of the teachers for the specific classrooms.

In order to cover longer distances *bonuses* are given to the school in various situations (i.e., walking or cycling with bad weather conditions). Furthermore, KGG also includes class-level and school-level *challenges*: upon completion of a specific objective (e.g., *zero emission day*, *no-car week*, etc.) the class/school benefits from virtual prizes that can be exploited in the journey to gain additional kilometers (i.e., a cruise ticket along a river, or a plane ticket to reach an oversea stop).

The journey progress is shown in the KGG Web application, and every time a virtual stop is reached, the teacher can use the associated multimedia educational material for in-class lessons.

B. The Play&Go Application

Play&Go (P&G) is a mobile application currently used by the citizens of Trento in Italy that fosters sustainable mobility behaviors. Citizens (i.e., players), can track their daily movement through the app by specifying whether they are walking, riding a bike or using a public transportation mean. At the end of the journey they receive an amount of *Green Leaves* points proportional to the amount of kilometers travelled and the level of sustainability of the used mean.

Players can monitor their history and achievements in their profile, where they can keep track of their progress by a variety of *badges* symbolizing particular achievements, such as reaching a certain amount of *Green Leaves*, or the usage of a specific transportation mean (e.g., an additional bike badge is assigned every 10 trips by bike), or the exploration of mobility alternatives (e.g., when using a designated Park&Ride facilities for the first time, i.e. parking lots with public transport connections, or exploring different Bike Sharing stations).

The game is structured in timeframes of one week each. At the beginning of each week, an email is sent to all participants presenting personalized *challenges*, which grant Green Leaves points, and announcing the weekly prizes. At the end of the week, physical prizes are assigned to top players and a communication is sent via mail with the recap of the week activity and the information about the winners. The reward for each challenge is also defined and takes into account the effort that the players are supposed to make to win it on the base of their skills.

Weekly and global *leaderboards* allow players to compare their performance to other players in terms of collected *Green*

[4]https://kidsgogreen.eu

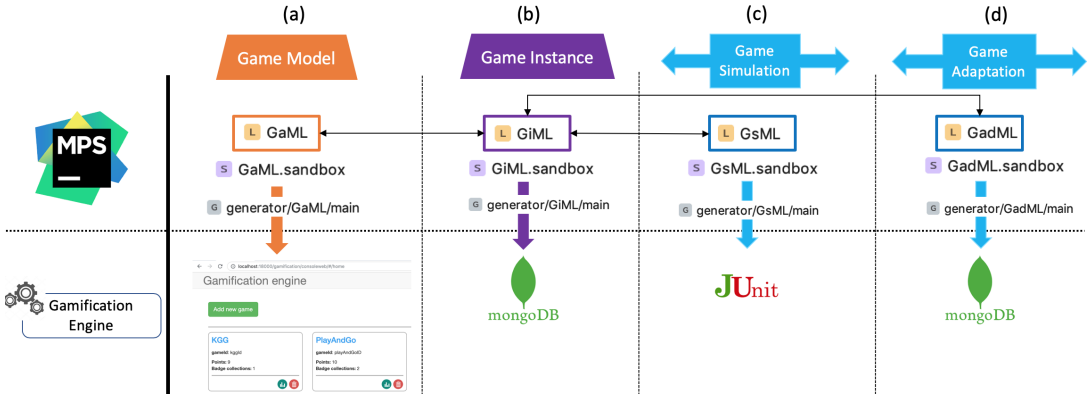

Fig. 13. The GDF Internal Logic.

Leaves, to motivate them to reach an even higher score. Furthermore, *leaderboards* promote social comparison, which can be an important provider of motivation.

C. GDF Technical Details

Figure 13 shows the internal logic of the GDF, made by four components: (a) Game Model, (b) Game Instance, (c) Game Simulation, and (d) Game Adaptation. For the interested reader, the prototype is available in its entirety on the GitHub repository defined for the framework. Each component is made-up of two layers, the former is implemented using MPS while the latter exploits an open-source Gamification Engine [26]–[28]. The engine has been realised in GitHub under the Apache License Version 2.0[5] and is available as stand-alone application as well as software-as-a-service (SaaS).

The *Game Model* component is composed by the *GaML* language devoted to the definition of a *Game* in a specific domain (e.g., Education or Mobility) and a *generator* able to transform this definition in a *Java* program used to deploy it in the gamification engine.

To realize the two applications described in Section V-A and Section V-B, we started by defining the set of *game elements* (i.e., actions, points, challenges, etc.), with their corresponding types (i.e., dataDrivenAction, eventDrivenAction, experiencePoint, singlePlayer challenge, etc.). By using these core elements, and thanks to the `GaML` language, we defined new games and deployed them in the gamification engine. The deployment step is done using the GaML generator described in Section IV-B and depicted in Figure 5. When executed, the generated Java code results in the creation of corresponding *game model*s.

The *Game Instance* component has been introduced to instantiate the different games starting from the same game Model. Starting from a *Game Instance* model, as the one in Figure 14, and using the generator introduced in Figure 7, we generated Java code as the one illustrated by Figure 15. This code is used to initialize the game execution environment composed by the set of *Teams* and their respective *Players* that

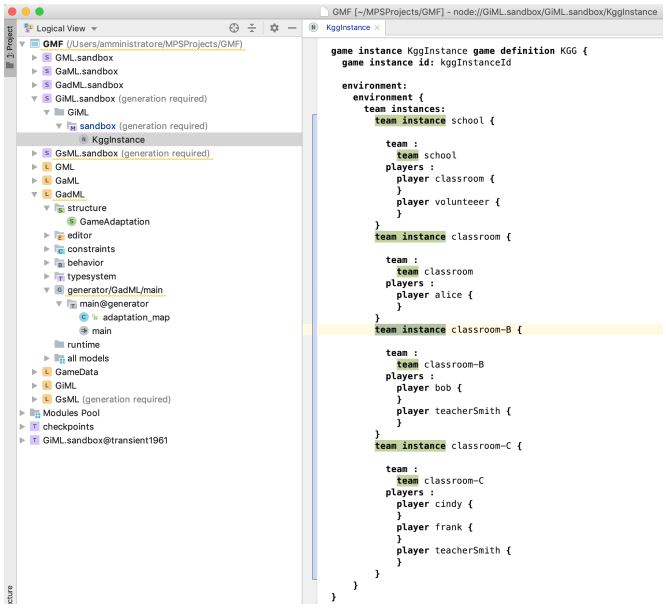

Fig. 14. Game Instance Model (KGG Application).

are ready to run the specific game instance. The execution of the code leads to the creation of a database that the gamification engine uses during the game execution. In particular, as depicted in Figure 13(b), a mongoDB[6] file is created.

To simulate the defined games, GDF uses the GsML component. As we have described in Section IV-D, a game simulation is defined by a *Team* and/or a *Player* that should execute an *actionInstance* or a *challengeInstance*. Starting from a *GameSimulation* model specified using GsML and containing the elements we mentioned earlier, we used the generator introduced in Figure 10 to derive the corresponding Java code. In this case a `jUnit` test, as the one depicted in Figure 16, is generated. In the specific case, it represents a game simulation of the P&G application where a specific player (i.e., prowler) executes two times the `Save_itinerary` action: the first time walking while the second time biking.

[5]https://github.com/smartcommunitylab/smartcampus.gamification

[6]https://www.mongodb.com/

```java
public class KGGInstance {


    @Autowired
    PlayerService playerSrv;

    private static final String GAME = "KGG";
    private static final String DOMAIN = "Education";



    public void initEnv() {
        TeamState team = new TeamState(GAME, "school");
        team.setName("school");
        team.setMembers(Arrays.asList("classroom", "volunteer"));
        playerSrv.saveTeam(team);

        team = new TeamState(GAME, "classroom");
        team.setName("classroom");
        team.setMembers(Arrays.asList("alice"));
        playerSrv.saveTeam(team);

        team = new TeamState(GAME, "classroom-B");
        team.setName("classroom-B");
        team.setMembers(Arrays.asList("bob", "teacherSmith"));
        playerSrv.saveTeam(team);

        team = new TeamState(GAME, "classroom-C");
        team.setName("classroom-C");
        team.setMembers(Arrays.asList("cindy", "frank", "teacherSmith"));
        playerSrv.saveTeam(team);


        // .....
    }
```

Fig. 15. Game Environment - generated class.

```java
public class PeriodicPointGameTest extends GameTest {

    @Autowired
    PlayerService playerSrv;

    private static final String GAME = "P&G";
    private static final String ACTION = "save_itinerary";
    private static final String DOMAIN = "Mobility";


    @Override
    public void defineExecData(List<ExecData> execList) {
        Map<String, Object> data = new HashMap<String, Object>();
        data.put("walkDistance", 1d);
        ExecData input = new ExecData(GAME, ACTION, "prowler", data);
        execList.add(input);

        DateTimeUtils
                .setCurrentMillisFixed(new GregorianCalendar(2016, 3, 26, 3, 22).getTimeInMillis());
        data = new HashMap<String, Object>();
        data.put("bikeDistance", 5d);
        input = new ExecData(GAME, ACTION, "prowler", data);
        execList.add(input);

    }

    // .....
```

Fig. 16. jUnit Test for the Game Simulation (P&G Application).

Finally, whenever a game adaptation (i.e., new challenge) must be executed for a specific player of team, we use the generator introduced in Figure 12. As in the case of the *Game Instance* component, the execution of the generated Java code leads to the creation of a Database (as depicted in Figure 13(c)) that the gamification engine uses to update the game instance of a specific player.

VI. Lessons Learned and Future Investigations

The framework illustrated in this paper can be considered a domain independent solution for the design of gamified applications. In fact, the architecture of GDF allows to define different gamification mechanisms and even to re-define how game elements should be combined. The sole aspect that should be considered as specific of the applications shown in this paper is the target gamification engine. Indeed, deploying the game on different target platforms would require the definition of new code generators. In this respect, the effort could vary from minimal to considerable, depending on whether the target language was Java-like or not, respectively. Previous experiences with similar modeling needs [29] made us immediately opt for a text-based solution; game definitions are essentially sets of rules and their compositions, and graphical concrete syntaxes tend to be intricate to use when combining more than two rules, possibly including negative application conditions. In this respect, the choice of MPS was mainly due to: (limited) previous knowledge of the language workbench; its projectional editors feature, inherited by any language defined through MPS; the native code generation support for Java.

The choice of a multi-level modelling approach came out directly from the nature of gamification applications: gamification principles, instantiated in terms of game elements, in turn materialised as game element instances [23]. As already mentioned throughout the paper, a traditional two-level modeling approach would have posed important issues in terms of maintenance and evolution of the framework. Notably, adding a new element in GaML would have required the verification of its correctness with respect to the elements existing in GML (i.e., its supertypes); moreover, the new added element would have needed apposite validation constraints to ensure that GiML elements connected to it as its instances were of the correct types. The solution discussed in this work leverages language imports and inheritance mechanisms provided by MPS to implement the desired GDF multi-layer architecture. As noticed already in [30], MPS cannot be considered a multi-level language workbench in a rigorous way; notably, there is no notion of *instantiation depth*, nor of *clabjects*, and so forth [5]. However, MPS smoothly supports language embedding through which it is possible to reuse/extend the concepts defined in one language for the specification of another language. In this respect, the number of levels and instantiation depth are determined dynamically by the use of a language. For example, GML can be placed at level M4, GaML at M3, GiML at M2, `Game` in GML has a depth of 1, being instantiated by `GameDefinition` in GaML, while `Challenge` has a depth of 2, being instantiated by concrete challenges in GiML or GsML. In our experience, these mechanisms make the definition of the different modelling levels and their interconnections by far a simpler task if compared to realising the same framework through a "proper" multi-level modelling approach [5].

From an end-user perspective, the typical starting point of traditional gamification development are graphical interfaces to define the skeleton of the application, i.e. core game components and some of the relationships among them, while large parts of the business logic are delegated to code snippets opportunely attached to the game elements. The abstraction gap introduced by code snippets has two main drawbacks: i) for domain experts it becomes difficult to keep track of lower level of abstraction details; ii) for programmers the complexity and error-proneness of the required code grows. Instead, by means of GDF and its sub-languages domain experts are able to handle details at lower levels of abstraction to the granularity of point update operations. Moreover, the quality of the applications is improved thanks to the checks

embedded in the language definitions, such that it would not be possible to misspell game elements, nor to define erroneous state values. These checks are in general difficult to perform when using code snippets, since the code is managed as text and is not knowledgeable of the concepts defined at higher levels of abstraction (notably challenges, players, etc.). The mentioned improvements become more evident when managing a running application: in fact, an important challenge in the development and evolution of gameful systems is the ability to revise or introduce new game elements and mechanics during the system execution, e.g., to keep the desired level of users' engagement. In this respect, GDF allows domain experts to reason about gamification specific concerns while implementation details are automatically handled by the code generators. Notably, experts might conceive new challenges and/or reward mechanisms, simulate them through GsML, and deploy them as adaptations by means of GadML. For these reasons and our personal experiences matured through the two case studies, we are confident about noticeable improvements in development time and application quality when adopting GDF as opposed to writing source code. Nonetheless, a more solid empirical evaluation about the proposed solution has still to be done, and is planned as future work.

Apart a thorough empirical validation of the proposed gamification modeling approach, we plan to investigate further adaptation and learning capabilities of GDF, to move from third-party to self-adaptation features [24], thus introducing more sophisticated self-adaptation mechanisms in gameful systems.

VII. Related Works

Gamification is gaining popularity in all those domains that would benefit from an increased engagement of their target users [1]. Therefore, gamification applications are found in disparate contexts, such as education [9], [31], health and environment [7], [25], e-banking [32], and even software engineering [3], just to mention a few. The growing adoption of gamified solutions made their design and development increasingly complex, due to e.g. the number and variety of users, and the mission criticality of some of the applications. Therefore, in general a rigorous development process is recommended to avoid the gamification project to fail [4]. In this respect, the approach proposed in this paper is not the first gamification design and deployment solution based on MDE. Notably, in [31] the authors propose a gamification framework targeting learning of modelling. The authors propose to partition the problem in similar sub-problems, like modelling the game, its main components, and the mechanics. Moreover, the authors provide automation in terms of listeners for detecting users' actions and corresponding achievements. Compared to our approach, the solution in [31] is domain-specific, and therefore does not require expressive power to define new game mechanisms, new game elements, and so forth. On the other hand, the domain-specificity allows the authors to implement automated monitors for detecting users' actions and keep track of the achievements. In this respect, we consider these automated trackings as target platform specific and not part of the game definition.

A domain independent solution is presented by Calderón et al. called MEdit4CEP-Gam [33]: it proposes a methodology and tool for defining gamified applications through complex event programming (CEP) and MDE. In particular, the approach exploits MDE techniques to define a game at a higher level of abstraction, and to generate corresponding code for a CEP engine. In this respect, the solution proposed in [33] is conceptually similar to the framework proposed in this paper. However, that solution exploits a two-layered metamodelling architecture, which in our opinion makes gamification design more intricate due to the flattening of different instantiation layers. Moreover, the tool is based on a diagrammatic concrete syntax, which in general does not scale well in case of complex rule definitions.

The market offers a plethora of other possible gamification resources, as presented and continuously updated in [34]. The main difference between the solutions proposed in the website and the approach proposed here is that the former ones are preconceived for a certain application domain, and very often the designer has little control about the game elements and mechanics.

As mentioned several times, the gamification framework illustrated in this paper is conceived on the basis of multi-level modelling (referred to also as deep metamodelling). A comparison between the available multi-level modelling alternatives goes beyond the scope of this work, and the reader is referred to [5] for an introductory discussion about existing solutions. Here we want just to remark that, to the best of our knowledge, there exist no gamification approaches based on multi-level modelling.

VIII. Conclusions

In this paper we presented the experiences matured in the development of a MDE approach for designing gameful systems. The proposed solution is based on multi-level modelling, such that the game definition is performed by successively refining the specification of the game, from the definition of main game components towards the instantiation of concrete game elements. The proposed layered architecture allows domain experts to abstract implementation problems and focus on details closer to their expertise, namely gamification techniques and application domain targeted by the game. The proposed mechanisms have been implemented by means of the MPS text-based language workbench. The resulting GDF not only reduces the complexity of defining gameful applications, but it also discloses the opportunities of specifying simulations and adaptations for particular gaming scenarios.

Acknowledgment

This work was partially funded by the EIT Climate KIC projects CLIMB Ferrara and SMASH.

References

[1] Jonna Koivisto and Juho Hamari. The rise of motivational information systems: A review of gamification research. *International Journal of Information Management*, 45:191 – 210, 2019.

[2] Mihaly Csikszentmihalyi and Isabella Selega Csikszentmihalyi. *Optimal experience: Psychological studies of flow in consciousness*. Cambridge university press, 1992.

[3] Oscar Pedreira, Félix García, Nieves Brisaboa, and Mario Piattini. Gamification in software engineering - a systematic mapping. *Information and Software Technology*, 57:157 – 168, 2015.

[4] Benedikt Morschheuser, Lobna Hassan, Karl Werder, and Juho Hamari. How to design gamification? a method for engineering gamified software. *Information and Software Technology*, 95:219 – 237, 2018.

[5] Juan De Lara, Esther Guerra, and Jesús Sánchez Cuadrado. When and how to use multilevel modelling. *ACM Trans. Softw. Eng. Methodol.*, 24(2):12:1–12:46, December 2014.

[6] Andrzej Marczewski. *Gamification: A Simple Introduction and a bit more - tips, advice and thoughts on gamification (2. ed)*. Andrzej Marczewski, 2012.

[7] Daniel Johnson, Sebastian Deterding, Kerri-Ann Kuhn, Aleksandra Staneva, Stoyan Stoyanov, and Leanne Hides. Gamification for health and wellbeing: A systematic review of the literature. *Internet Interventions*, 6:89–106, 2016.

[8] Richard M Ryan, C Scott Rigby, and Andrew Przybylski. The motivational pull of video games: A self-determination theory approach. *Motivation and emotion*, 30(4):344–360, 2006.

[9] Darina Dicheva, Christo Dichev, Keith Irwin, Elva J. Jones, Lillian (Boots) Cassel, and Peter J. Clarke. Can game elements make computer science courses more attractive? In *Proceedings of the 50th ACM Technical Symposium on Computer Science Education, SIGCSE 2019*, page 1245, 2019.

[10] Sebastian Deterding, Dan Dixon, Rilla Khaled, and Lennart E. Nacke. From game design elements to gamefulness: defining "gamification". In *Proceedings of the 15th International Academic MindTrek Conference: Envisioning Future Media Environments, MindTrek 2011*, pages 9–15, 2011.

[11] Katie Salen and Eric Zimmerman. *Rules of play: game design fundamentals*. MIT Press, 2004.

[12] Gustavo Fortes Tondello, Hardy Premsukh, and Lennart E. Nacke. A theory of gamification principles through goal-setting theory. In *51st Hawaii International Conference on System Sciences, HICSS 2018*, 2018.

[13] Manuel Schmidt-Kraepelin, Scott Thiebes, Minh Chau Tran, and Ali Sunyaev. What's in the game? developing a taxonomy of gamification concepts for health apps. In *51st Hawaii International Conference on System Sciences, HICSS 2018*, pages 1–10, 2018.

[14] Kevin Werbach and Dan Hunter. *The gamification toolkit : dynamics, mechanics, and components for the win*. Wharton Digital Press, Philadelphia, 2015.

[15] Martin Ward. Language oriented programming. *Software-Concepts&Tools*, 15:147–161, 01 1994.

[16] Markus Voelter and Sascha Lisson. Supporting diverse notations in mps' projectional editor. In *Proceedings of the 2nd International Workshop on The Globalization of Modeling Languages co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages and Systems, GEMOC@Models 2014*, pages 7–16, 2014.

[17] Markus Voelter. Language and IDE modularization and composition with MPS. In *Generative and Transformational Techniques in Software Engineering IV, International Summer School, GTTSE 2011*, pages 383–430, 2011.

[18] Markus Voelter and Vaclav Pech. Language modularity with the MPS language workbench. In *34th International Conference on Software Engineering, ICSE 2012*, pages 1449–1450, 2012.

[19] Markus Voelter, Sebastian Benz, Christian Dietrich, Birgit Engelmann, Mats Helander, Lennart C. L. Kats, Eelco Visser, and Guido Wachsmuth. *DSL Engineering - Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.

[20] Fabien Campagne and Fabien Campagne. *The MPS Language Workbench, Vol. 1*. CreateSpace Independent Publishing Platform, USA, 1st edition, 2014.

[21] Fabien Campagne. *The MPS Language Workbench Volume II: The Meta Programming System (Volume 2)*. CreateSpace Independent Publishing Platform, USA, 1st edition, 2016.

[22] Robert France and Bernhard Rumpe. Model-based development. *Software and Systems Modeling*, 7(1):1–2, 2008.

[23] Colin Atkinson and Thomas Kühne. Model-driven development: A metamodeling foundation. *IEEE Softw.*, 20(5):36–41, September 2003.

[24] Franck Barbier, Eric Cariou, Olivier Le Goaer, and Samson Pierre. Software adaptation: Classification and a case study with state chart xml. *IEEE Software*, 32(5):68–76, Sep. 2015.

[25] Annapaola Marconi, Gianluca Schiavo, Massimo Zancanaro, Giuseppe Valetto, and Marco Pistore. Exploring the world through small green steps: improving sustainable school transportation with a game-based learning interface. In *Proceedings of the 2018 International Conference on Advanced Visual Interfaces, AVI 2018*, pages 24:1–24:9, 2018.

[26] Raman Kazhamiakin, Annapaola Marconi, Mirko Perillo, Marco Pistore, Giuseppe Valetto, Luca Piras, Francesco Avesani, and Nicola Perri. Using gamification to incentivize sustainable urban mobility. In *IEEE First International Smart Cities Conference, ISC2 2015, Guadalajara, Mexico, October 25-28, 2015*, pages 1–6, 2015.

[27] Raman Kazhamiakin, Annapaola Marconi, Alberto Martinelli, Marco Pistore, and Giuseppe Valetto. A gamification framework for the long-term engagement of smart citizens. In *IEEE International Smart Cities Conference, ISC2 2016*, pages 1–7, 2016.

[28] Reza Khoshkangini, Giuseppe Valetto, and Annapaola Marconi. Generating personalized challenges to enhance the persuasive power of gamification. In *Proceedings of the Second International Workshop on Personalization in Persuasive Technology co-located with the 12th International Conference on Persuasive Technology, PPT@PERSUASIVE 2017*, pages 70–83, 2017.

[29] Antonio Bucchiarone and Antonio Cicchetti. A model-driven solution to support smart mobility planning. In *Proceedings of the 21th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems*, MODELS '18, pages 123–132, New York, NY, USA, 2018. ACM.

[30] Andreas Prinz. Multi-level language descriptions. In *Procs. of the 3rd Intl. Workshop on Multi-Level Modelling at MoDELS 2016*, pages 56–65. CEUR-WS, 2016.

[31] Valerio Cosentino, Sébastien Gérard, and Jordi Cabot. A model-based approach to gamify the learning of modeling. In *Proceedings of the 5th Symposium on Conceptual Modeling Education and the 2nd International iStar Teaching Workshop co-located with the 36th International Conference on Conceptual Modeling (ER 2017), Valencia, Spain, November 6-9, 2017.*, pages 15–24, 2017.

[32] Luís Filipe Rodrigues, Carlos J. Costa, and Abílio Oliveira. Gamification: A framework for designing software in e-banking. *Computers in Human Behavior*, 62:620 – 634, 2016.

[33] Alejandro Calderón, Juan Boubeta-Puig, and Mercedes Ruiz. Medit4cep-gam: A model-driven approach for user-friendly gamification design, monitoring and code generation in cep-based systems. *Information & Software Technology*, 95:238–264, 2018.

[34] Compare 120+ gamification platforms. https://technologyadvice.com/gamification/. Accessed: April 2019.