Enhancing Gameful Systems with a Domain Specific Language for Rules Lifecycle Management

Antonio Bucchiarone
MoDiS Research Unit
Fondazione Bruno Kessler (FBK)
Trento, Italy
bucchiarone@fbk.eu

Stefano Martella
Department of Computer Science
Information Engineering and Mathematics
University of L'Aquila, Italy
stefano.martella9614@gmail.com

Mario Fusco
Red Hat Italy
mfusco@redhat.com

Henry Muccini
Department of Computer Science
Information Engineering and Mathematics
University of L'Aquila, Italy
henry.muccini@univaq.it

***Abstract*—*Gamification* refers to the application of gaming concepts in non-playful contexts to encourage engagement with a product or service. *Gameful systems* are software programs that embed this concept and are increasingly being applied in various fields, such as environmental awareness, education and training, health, and food waste. These systems are based on a set of *rules* defined by game designers, who specify the conditions that must be met to evolve the state of a game and the associated consequences. However, the current approaches to defining rules in gameful systems carry certain limitations that can give rise to undesirable or unforeseen behaviors. These limitations pose significant challenges to the smooth functioning and overall effectiveness of gameful experiences. This paper presents a *Domain Specific Language (DSL)* designed to simplify and strengthen the entire rule lifecycle process. The DSL guides designers during the rule definition process using easy-to-use fluent APIs. Additionally, it enables the simulation of rules and verification of their behavior through a graph that explicitly shows their interconnection. Finally, the DSL facilitates the deployment of rules on the *gamification engine* with simple steps. This approach enhances the effectiveness of gameful systems by reducing the likelihood of unintended consequences and improving the overall user experience.**

***Index Terms*—Domain Specific Language, Gamification, Drools, Game Rules, Simulation, Deployment**

I. Introduction and Motivations

Gamification has been increasingly recognized as a promising approach to promoting sustainable behavior and raising environmental awareness [1], [2]. By integrating game-like elements into sustainability initiatives, such as energy conservation, waste reduction, and eco-friendly transportation, gamification can incentivize and motivate individuals to adopt more sustainable habits and lifestyles. Gamification can also foster social interaction, collaboration, and competition, which can enhance the sense of community and collective action towards sustainability. In this way, gamification can help address the complex and interconnected challenges of sustainability by engaging, educating, and empowering individuals to take action and make a positive impact on the environment [3].

Rules play a critical role in gamification as they define the boundaries, constraints, and objectives of the game [4], [5]. By establishing clear and compelling rules, game designers can guide the players' behavior, motivate them to take specific actions, and provide feedback on their performance. Rules also help create a sense of fairness, transparency, and predictability, which are essential for building trust and engagement among the players. Moreover, rules can be customized and adapted to different contexts, audiences, and goals, making gamification a flexible and versatile approach to designing interactive experiences. Therefore, it is essential to invest time and effort in defining rules that align with the desired outcomes and resonate with the players' interests and preferences [6]–[8].

The rule definition phase is a crucial aspect while designing gameful systems; during this process the game designers leverage on the so-called *gamification elements* (such as the concepts of *points*, *levels*, *leaderboards*, and so on) to specify the conditions that must be met to evolve the state of the game as well as the actual consequences on the state itself.

One of the most adopted solutions when it comes to defining rules is Drools [9], [10], an open source rule engine based on reactive computing models that allow defining the rules as two logical blocks: the *Left Hand Side* that describes the conditions to met for their execution and the *Right Hand Side* that corresponds to the actual consequences of the rules on the state of the game. Drools has its syntax through which is possible to define rules like the following:

Listing 1: Drools rule.

```
// The name of the rule.
rule "Underage"
  // Definition of the facts that trigger the rule;
  // in this case the rule is executed when a
       LoanApplication
  // and and Applicant whose age is less than 21 are
       present.
  when
    $application : LoanApplication()
```

```
    Applicant( age < 21, $name: name )
  // Consequences of the rule;
  // in this case the loan application is denied and an
       explanation
  // is added to the relative object.
  then
    $application.setApproved(false);
    $application.setExplanation( "Underage" );
end
```

Drools is a really powerful tool that permits the definition of complex scenarios for each rule. Nonetheless, its introduction brings a higher level of complexity that may lead to several limitations [11]: game designers are typically non-programmers that during the rules definition process are required to write code; thus, there is an important gap between their domain's knowledge and their coding skills. Moreover, the rules definition is not followed by a well structured testing phase that allows a prior verification of their semantic correctness, this means that some rules don't work as expected resulting in users dissatisfaction and complaints. Furthermore, in more complex games, rules might be interfering each other and result in unexpected behaviours; even in this case, the designers don't have an organized procedure to analyze the rules correlation and their possible collision.

Using a domain-specific language (DSL) to specify gamification rules can provide several advantages over using a more general-purpose language. A DSL is a language designed to address the specific needs and requirements of a particular domain, in this case, gamification [12], [13].

One of the primary benefits of using a DSL is that it can make it easier to express complex ideas and concepts in a concise and intuitive way [14], [15]. By using specialized terminology and syntax, a DSL can capture the nuances and subtleties of gamification in a way that would be difficult or cumbersome to express in a more general-purpose language.

Another advantage of using a DSL is that it can make it easier to validate and verify gamification rules. A well-designed DSL can include built-in checks and constraints that ensure that the rules are consistent and coherent, and that they conform to best practices and industry standards. This can help to reduce errors and inconsistencies, and improve the overall quality of the gamification implementation.

Finally, using a DSL can also make it easier to customize and extend gamification rules over time. Because a DSL is designed specifically for the domain of gamification, it can be more modular and extensible than a general-purpose language. This can make it easier to modify and update the rules as needed, without having to rewrite large portions of the code or risk introducing new bugs or errors.

In summary, using a domain-specific language to specify gamification rules can provide a range of benefits, including improved expressiveness, better validation and verification, and greater flexibility and extensibility.

The objective of this paper is to present how a *Domain Specific Language* [16] [17] [18] can be designed and implemented in order to exploit the mentioned advantages to overcome the actual limitations and to offer a completely rethought rule's lifecycle for the designers to provide an overall enhanced experience.

In particular, **DSL4GaR** (**D**omain **S**pecific **L**anguage for **Ga**mification **R**ules) aims to supply **an easy interface for the rules definition** also enriched with syntax highlight and auto-completion mechanisms to provide a higher level of abstraction to the code itself for the designers. Furthermore, the DSL allows to generate **a graph representing the evolution of the state of a game** based on the executed rules; this permits on the one hand to verify the correctness of the defined rules and on the other hand to verify their possible interference. Moreover, the DSL allows the **deployment of the generated rules** directly inside the gamification engine. Thus, leveraging on the DSL, the designers can handle the whole process without leaving the environment they are working in.

II. Related Works

In the literature we found different approaches that exploit the DSLs for designing and deploying gameful systems. One of the first attempt to formalize a domain specific language for gamification concepts is represented by the Gamification Modelling Language (GaML) [19]. It consists of a rigorous syntax designed to be readable and writable by gamification experts and to ease the communication among the various stakeholders of a gameful system.

A further evolution of the GaML can be found in the *Gamification Design Framework* (GDF) [20]. It is a multi-level modelling approach for the characterization of gameful systems. In particular, it provides a projectional DSL for the definition of each concern (game elements, game mechanisms and game dynamics) to automate the corresponding code generation phase. Each DSL is implemented by means of JetBrains Meta Programming System (MPS)[1], a text-based meta-modelling framework. With this approach, designers can incrementally build gameful systems by leveraging on each of this languages. The GDF also provides a further DSL for handling the simulation phase of the defined gameful systems.

MEdit4CEP-Gam [21] is another example of a model-driven approach for user-friendly gamification design, monitoring and code generation. In this case, the text-based approach is discarded in favour of a graphical DSL to formalize gamification designs.

Although the described solutions provide powerful tools for the designers, they also present drawbacks. On the one hand, GDF allows the definition of the rules by means of MPS which syntax is not as simple as other options to learn for non-programmers people. On the other hand, MEdit4CEP-Gam eases the design process through a graphical interface but it does not provide a way for the examination of *chained* executions of the rules and their possible interference. In the proposed DSL the idea is to combine the the advantages of both approaches to harden the whole rule's lifecycle process. In particular, the DSL provides a set of Java *fluent* APIs for designing the rules without constraining the possible scenario

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

to represent but, at the same time, guiding the designers during the process to avoid errors. Furthermore, the DSL must provide an effective tool to permit rules simulation, thus to analyze the order of their execution and to verify the actual changes to the state brought by each one of them; this would enable the designers to have a deep understanding of the impact of each rule on the game and to identify potential malfunctioning when one or more rule are triggered, for example, by the same facts.

The DSL allows to perform such analysis through a graphical interface reporting the exact timeline of the rules, their consequences on the state of the game and eventually the errors during their execution.

Fig. 1: Game state transitions.

III. DSL4GaR Functionalities

DSL4GaR serves as a tool to raise the level of abstraction for non-technical people during the whole rule's lifecycle process. In particular, it aims to provide a set of easy-to-use APIs through which it is possible to define the rules inside an environment (i.e., integrated development environment (IDE)) that guarantees syntax highlights, auto-completion mechanisms and, at the same time, that constraints the user inputs to reduce possible errors. Specifically, game designers are guided by the IDE and its support features during the whole design phase, thus, during the definition of both the LHS and RHS of the rules; this allows the designer to focus on the semantics of the rules rather than their syntax.

DSL4GaR also supplies a well-structured mechanism to test the rules and verify their outcomes; the designers can easily define a game's initial state, through the provided APIs, and simulate the execution of the rules to monitor how the state (or part of it) evolves as well as rules interaction through a graph (figure 1):

Each node of the graph contains a state of the game while each edge represents the executed rule that evolves the system to a new state. During the simulation the designers can also define a list of assertions on the game's state that are automatically verified during the process; if something unexpected happens, the graph reports the encountered errors and the relative unverified assertions (figure 2); this allows the game designer top suddenly identify the rules causing the error and to investigate them.

Fig. 2: Assertion error.

Moreover, the graph's nodes can be either filtered by selecting the gamification elements of interest or by specifying a keyword (figure 3); in this way, the analysis process is made smarter and smoother.

Fig. 3: Graph filtered for badge collection concept.

Once the rules are defined and tested, they are ready to be

deployed on the gamification engine (GE) [22], the software that takes care of their execution allowing the final user to interact with the gameful system; in this case the DSL4GaR allows a direct interaction with the GE in order to deploy, update or delete the rules. Even in this case, the DSL simplifies the process by providing a set of easy-to-use fluent API to perform the described operations.

IV. DSL4GaR Implementation

The DSL4GaR software architecture is composed of three primary modules, which are visually represented in Figure 4 along with their connections.

In addition to the three primary modules, the DSL4GaR software architecture is integrated with a gamification engine.

In the following, we will introduce each of these components and explore how they interact with each other.

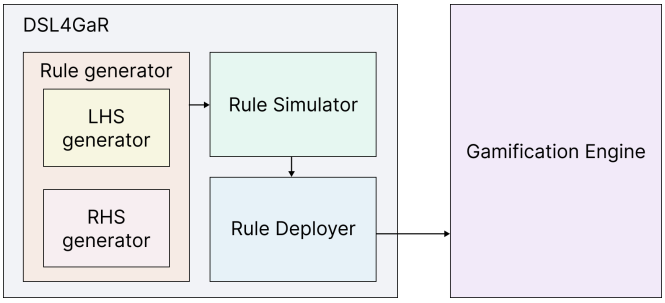

Fig. 4: DSL4GaR - Software Architecture.

- **Rule Generator**: it exposes a set of APIs useful to define a rule; it can be further decomposed into two sub-modules:
  - **LHS generator**: A set of APIs to properly define the *conditions* for a rule to be executed;
  - **RHS generator**: A set of APIs to build the *consequences* of a rule when it is executed.
- **Rule Simulator**: it exposes a set of APIs used to simulate the defined rules. Each simulation is composed by an initial state, the set of rules applied, and all the state evolutions obtained during the simulation execution. The state evolution is represented by a graph that is used to verify the impact of each applied rule in the simulation.
- **Rule Deployer**: through it is possible to interact with the gamification engine and deploy all the tested rules after their simulation.

Each module is designed to guarantee *adaptability* of the DSL with the various rules engines; this means that potentially not only Drools rules can be generated but also other engine artifacts (where the rules are expressed as logic blocks) could be produced. Moreover, DSL4GaR has been implemented bearing in mind *extensibility*; the addition of new gamification elements is a pretty straightforward process.

DSL4GaR extensively relies on Java interfaces and generic types; each gamification element is represented by an interface and its implementation that are embedded in the *LHS generator* module. To allow the DSL to support a new gamification element it is sufficient to define a new interface (that describes the element's attributes and its possible constraints) and the corresponding implementation in that module. This is possible thanks to the usage of the *Apache Velocity*[2]. With this tool is possible create reusable *templates* that can be used across multiple gamification rules, which can save time and effort when building complex gameful systems. The engine uses a simple yet powerful syntax that allows for the manipulation of variables, loops, and conditions within the templates, making it a flexible and versatile tool for development. Overall, Apache Velocity is a valuable tool for any developer looking to streamline the development process and improve the maintainability and scalability of their applications. In this way also the addition of a RHS template is a straightforward process: it is sufficient to add a new template that describes the consequence of a rule and its implementation in the *RHS generator* module. Moreover, thanks to its structure, DSL4GaR can be extended to not only support Drools but potentially every rule engine that describes the rules as two building blocks (LHS and RHS); even in this case, this is achievable by adding a new implementation for the interfaces based on the supported language. Thus, by using the same APIs, the generated code can either be DRL or potentially any other notation.

The DSL has been implemented in Java for two main reasons: (1) it is a widely used language easy to understand (at least for the needed subset of functionalities) and with a large variety of IDE with integrated support, a fundamental aspect for text-based languages; (2) The rules taken into account for the purpose of this analysis are expressed as DRL (Drools rule), this means they already embed Java code in their RHS.

The main idea behind DSL4GaR is to provide a higher level of abstraction concerning the code; for this reason, all the APIs exposed for handling the various phases of the rule's lifecycle are designed leveraging the *Fluent Interface* [23] pattern; this allows to have a code that can be read/written nearly as human language. In particular, for the LHS generation of each gamification element, a correspondent method to fluently constraint its fields has been provided; instead, for the RHS generation a further step has been added with the introduction of Apache Velocity engine[3]. Through this engine it is possible to define Java code templates (.vm files) enriched with placeholders like the following:

```
System.out.println("${message}");
```

Then, by defining a context, it is possible to specify the proper values for the placeholders to be substituted inside the templates as shown in listing 2.

Listing 2: Placeholder replacement.

```
VelocityEngine velocityEngine = new VelocityEngine();
Template template = velocityEngine.getTemplate("
    printMessage.vm"); // from resources
```

[2]https://velocity.apache.org/

[3]https://velocity.apache.org/engine/

```
VelocityContext velocityContext = new VelocityContext();
velocityContext.put("message", "Hello World!");

StringWriter stringWriter = new StringWriter();

template.merge(velocityContext, stringWriter);
```

The substitution is carried by the Velocity engine itself. Thus, a set of possible consequences of a rule have been extracted from a list of rules currently used by gameful systems to generate a set of Velocity templates. Such templates allow to define RHS that are useful to assign badges, mark challenges as completed, assign levels to players, update players' scores, and so forth.

An example of a rule definition is provided in listing 3.

Listing 3: Rule definition.

```
final ChallengeBind CHALLENGE_BIND = new ChallengeBind("
    challenge");
final PointBind POINT_BIND = new PointBind("pc");

return new PackageDescrBuilderImpl()
        .name("eu.trentorise.game.model")
        .newImport(ChallengeConcept.class.getName()).end()
        .newImport(PointConcept.class.getName()).end()
        .newImport(PointConcept.class.getName()).end()
        .newRule()
        .name("challenge")
            .attribute("lock-on-active", "true")
            .when()
                .challenge(CHALLENGE_BIND).modelName(EQ,
                    Challenge.PRIZE).end()
                .point(POINT_BIND).name(EQ, Point.
                    GREEN_LEAVES).end()
            .end()
            .then()
                .completeChallenge(CHALLENGE_BIND)
                .incrementScore(POINT_BIND, 100d)
            .end()
        .end()
        .getDescr();
```

The above listing produces the DRL code reported in listing 4.

Listing 4: Corresponding DRL generated from listing 3.

```
package eu.trentorise.game.model

import eu.trentorise.game.model.ChallengeConcept
import eu.trentorise.game.model.PointConcept
import eu.trentorise.game.model.PointConcept

rule "challenge"
    lock-on-active true
when
    $challenge : ChallengeConcept( modelName == "prize" )
    $pc : PointConcept( name == "green leaves" )
then
    $challenge.completed();
    update($challenge);
    $pc.increment(100.0);
    update($pc);
end
```

The Fluent Interface pattern has been also used for the implementation of both the simulation and the deployment modules to guarantee an high level **adaptability of the DSL**; since the adopted pattern leverages on interfaces, it is possible to have multiple implementations that allow code generation not only for the DRL (Drools rules) but potentially for any other technology where rules are expressed as when/then blocks. Furthermore the DSL has been implemented bearing in mind its **extensibility**; the addition of a new gamification element is a pretty straightforward operation.

V. Illustrative Example

Ferrara Play&Go [24] [4] is a an example of gameful system which aims to make the use of sustainable means of transportation enjoyable and rewarding [25]. The game is available for every citizen of an Italian city, Ferrara, and allows them to track their journey using a mobile application (from Figure 5 to Figure 8). The supported transportation means are bike, bus, train or walking; each trip makes players earn *Green Leaves* points based on the length of the travelled path and the sustainability of the used transportation means. Each week, players are ranked through leaderboards where the most performant are rewarded with badges and points as well as with real prizes.

Overall score is considered alongside with weekly score in order to ensure fairness both for the late-comers and the abitudinary players. To further motivate and retain players, the game provides weekly challenges both in single-player and 2-players mode. To avoid overwhelming new users with a multitude of game mechanisms, Ferrara Play&Go includes a leveling system that enables to unlock challenges only when a certain levels are reached (e.g. from level 4 the first type of multiplayer challenge is available).

In the 2022 edition, 2197 citizens have joined the initiative, with more than 58.000 trips tracked and more than 249.000 sustainable kilometers traveled. Play&Go has shown both the ability of the approach to support citizen participation in long-term games, and the ability to break habits and change the behavior of players towards more sustainable mobility habits [25] [26].

A. Rules lifecycle

The rules behind *Ferrara Play&Go* application were defined without the DSL. The objective of the reported case study is to highlight the advantages introduced by the DSL and to show how the rules design workflow is changed. In particular, the DSL allows to handle the whole lifecycle of the rules inside the same environment (IDE), starting from the generation, through simulation to deployment [5]. The main game concepts adopted for the definition of the rules of Ferrara Play&Go are reported below:

- **Points:** They represent a reward for the successful accomplishment of in-game activities and they are expressed as numerical values. Players can earn points of different natures (e.g., Bike Km, Walk Km, Bus Trips and so on).
- **Badge Collections and Badges:** The app includes a large number of achievements that players can unlock while using it; a badge embeds the visual representation

[4] https://playngo.it/

[5] https://stefanomartella.github.io/DSL4GaR/docs/playground/ferrara-play-and-go

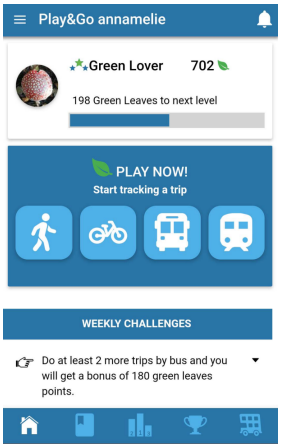

Fig. 5: Tracking Progress.

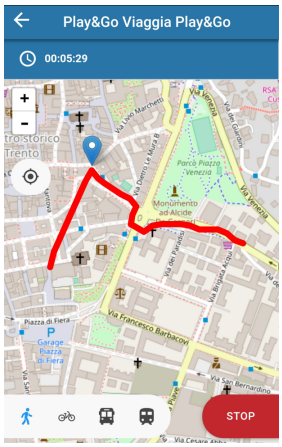

Fig. 6: Path tracking.

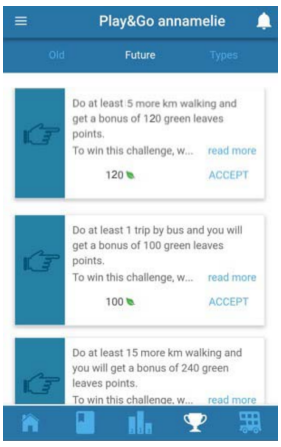

Fig. 7: List of challenges.

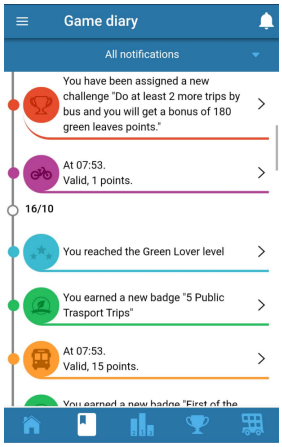

Fig. 8: Progresses list.

of these achievements. Badges are organized in Badge Collections, each symbolizing a player's performance in a specific area.

- **Levels:** Levels provide a feeling of progress and mastery to the players; each level is associated with a specific Point Concept. Every time a user earns enough points, he reaches a new level that eventually unlocks new challenges.
- **Leaderboards:** Even if motivational potential of leaderboards is debatable [27] [28], they are a really useful game concept to compare users performances each other and to stimulate competition dynamic among them. In Ferrara Play&Go this concept is used to rank players based on different criteria (e.g., type of points, weekly achievements and so on).
- **Challenges:** Challenges require players to fulfill a specific goal whose attainment requires an individual commitment for a limited period of time. For each completed challenge, the user is rewarded with points or badges (i.e. "Do at least 5 km walking and get a bonus of 120 green leaves"). The objective of this game concept it to keep players in a state of flow, where the player is fully engaged with the system [29]. It is important to notice that each challenge must be personally calibrated for each player to prevent boredom or, on the opposite, anxiety and frustration [30].

1) Rule generation: Ferrara Play&Go presents a large set of rules that can be clustered as follow:

- **Point related rules:** these rules are triggered when the player performs a specific action and result in an overall increase of accumulated points for the player itself. Point's nature changes based on the type of action carried on by the player.
- **Badge related rules:** rules that assign badges to the players based on their points and the nature of the points themselves.
- **Classification related rules:** these rules are useful to rank the players through leaderboards based on the badges they collect over time.

Examples of rules belonging to the badge assignment cluster are the following:

Listing 5: Ferrara Play&Go DRL rules for badge assignment.

```
rule "10 point green badge"
    salience -1000
when
    PointConcept(name == "green leaves", score >= 10.0)
    $bc : BadgeCollectionConcept(name == "green leaves",
        badgeEarned not contains "10-point-green")
    Game( $gameId: id)
    Player( $playerId : id)
then
    $bc.getBadgeEarned().add("10-point-green");
    insert( new BadgeNotification($gameId,$playerId,"10-
        point-green"));
    update( $bc );
end

rule "50 point green badge"
    salience -1000
when
    PointConcept(name == "green leaves", score >= 50.0)
    $bc : BadgeCollectionConcept(name == "green leaves",
        badgeEarned not contains "50-point-green")
    Game( $gameId: id)
    Player( $playerId : id)
then
    $bc.getBadgeEarned().add("50-point-green");
    insert(new BadgeNotification($gameId,$playerId,"50-
        point-green"));
    update( $bc );
end
```

It is important to notice that in each cluster, the rules share a large portion of their codebase; this means, as shown in the above DRLs, that the rules to assign a "10-points-green-badge" or a "50-points-green-badge" badge to a player differ only for the amount of points the player is required to have but not for their main logic (at the end of the day the players earns a badge). This of course open the doors for reusability; in particular, the DSL allows to define *templates* that provide the basic structure of the rules with the same logic that can be re-used to build different rules of the same nature. In the following code snippet it is shown how, through the DSL, it has been possible to define a template for the rules in 5.

Listing 6: Badge template defined with the DSL

```
public static PackageDescr getBadgeTemplate(
        String ruleName,
        String pointName,
        String badgeCollectionName,
        String badgeName,
        Double scoreThreshold
) {
    final Bind GAME_ID_BIND = new Bind("gameId");
    final Bind PLAYER_ID_BIND = new Bind("playerId");
    final BadgeCollectionBind BADGE_COLLECTION_BIND = new
         BadgeCollectionBind("bc");

    return new PackageDescrBuilderImpl()
        .name("eu.trentorise.game.model")
        // Imports omitted
        .newRule()
            .name(ruleName)
            .attribute("salience", "-1000")
            .when()
                .point()
                    .name(EQ, pointName)
                    .score(GTE,scoreThreshold)
                .end()
                .badgeCollection(BADGE_COLLECTION_BIND)
                    .name(EQ, badgeCollectionName)
                    .badgeEarnedNotContains(badgeName)
                .end()
                .game().bindId(GAME_ID_BIND).end()
                .player().bindId(PLAYER_ID_BIND).end()
            .end()
            .then()
                .addBadgeWithNotification(
                     BADGE_COLLECTION_BIND, GAME_ID_BIND,
                     PLAYER_ID_BIND, badgeName)
            .end()
        .end()
        .getDescr();
}
```

During the template definition the designer is guided by the IDE through syntax highlight and autocomplete mechanisms that wouldn't have without the DSL; he is constrained by Java in the values he can type and the whole process is fluent thanks to provided APIs. The defined template (6) can then be used to reproduce the rules in 5.

Listing 7: Badge template usage

```
public static PackageDescr getGreenBadge10Rule() {
    return GenericBadgeRuleTemplate.getBadgeTemplate(
        "10 point green badge",
        Point.GREEN_LEAVES,
        BadgeCollection.GREEN_LEAVES,
        Badge.GREEN_10_POINTS,
        10.0
    );
}

public static PackageDescr getGreenBadge50Rule() {
    return GenericBadgeRuleTemplate.getBadgeTemplate(
        "50 point green badge",
        Point.GREEN_LEAVES,
        BadgeCollection.GREEN_LEAVES,
        Badge.GREEN_50_POINTS,
        50.0
    );
}

// Other rules based on the badge template
```

In Listing 7 for each of the rules based on the template, it is sufficient to pass the actual values to substitute the template's placeholders. The possibility to reuse code through templates increases the portability, maintainability and sustainability of the rules; this means that if the logic to assign badges evolves, it is no longer necessary to update each rule individually but it is possible to intervene only on the template instead.
In the case study, for each of the rules cluster defined above, a template has been created to define the associated rules[6].

2) Rule testing: With the current approach the game designers ship the defined rules directly on the gamification engine. The absence of a well structured testing phase before the deployment causes users dissatisfaction and complains when something unexpected happens (i.e. points or badges not gained). Currently, when a user notifies an unexpected behaviour, the related rule is analyzed to understand the cause of the malfunctioning; in particular, the designers try to replicate the state of the player that notifies the problem and simulate the execution of the rule. This whole process is totally dependant to the gamification engine, the state of the player is retrieved from there and the designers have a really hard time to represent all the elements needed for the rule. Moreover, the designers don't have a tool to study the correlation among the rules and the outcomes of their possible interference. The DSL offers the possibility to test the rules in a completely detached environment before their deployment and provides a set of APIs to define all the elements needed for the rule as well as a graphical visualization that represent the game's state evolution.

More specifically, the rule reported in Listing 7 can be tested through the DSL by defining facts in the following way:

Listing 8: Rule facts definition

```
// For "10 point green badge" rule
final PointFactBuilderImpl pointFact = PointFactBuilderImpl
     .builder().name(Point.GREEN_LEAVES).score(10.0).build()
     ;
final GameFactBuilderImpl gameFact = GameFactBuilderImpl.
     builder().id("game1").build();
final PlayerFactBuilderImpl playerFact =
     PlayerFactBuilderImpl.builder().id("player1").build();
final BadgeCollectionFactBuilderImpl badgeCollectionFact =
     BadgeCollectionFactBuilderImpl.builder()
        .name(BadgeCollection.GREEN_LEAVES)
        .build();
```

Since the facts are defined in order to trigger the rule, the expected output is to have the BadgeCollection fact to contain the "10-point-green" badge once the rule gets executed, thus:

Listing 9: Assertion on badge

```
CheckExpectationLambda doesContain10PointGreenBadge = () ->
      Assert.assertTrue(
                "Contiene il badge '10-point-green'",
                     badgeCollectionFact.getBadgeEarned().
                     contains(Badge.GREEN_10_POINTS));
```

If the expectation is verified successfully, the DSL outputs a graphical representation of the state that shows how it evolves for each rule execution (in this case one).

[6]https://github.com/StefanoMartella/FerraraPlayAndGo_GRL/tree/main/src/main/java/game/rules/templates

Fig. 9: Simulation output.

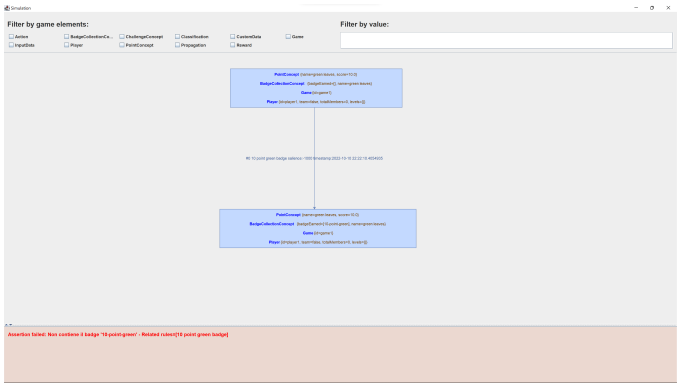

Fig. 10: Simulation output error notification.

As shown in Figure 9, after the rule "10 points green badge" gets executed the BadgeCollectionConcept inside the working memory is enriched with the new badge.

If the expectation would have been reverted as follow:

Listing 10: Assertion on badge reverted

```
CheckExpectationLambda doesContain10PointGreenBadge = () ->
     Assert.assertFalse(
CheckExpectationLambda doesContain10PointGreenBadge = () ->
     Assert.assertFalse(
               "Non contiene il badge '10-point-green'",
                    badgeCollectionFact.getBadgeEarned().
                    contains(Badge.GREEN_10_POINTS));
```

the DSL, through the graphical interface, would have notified the designer:

Of course, the DSL shows its true effectiveness when more rules are chained; in this case the designer can clearly follow the rules execution flow as well as the state updates for each one of them to have a full grasp of what is happening.

B. Rule deployment

After the testing phase, all the checked rules can be deployed directly on the gamification engine [7], usually a rule execution system capable of running a set of rules containing serious game logic. Rules can be added, updated or removed from the engine in order to evolve the mechanisms of the games. DSL4GaR allows the designers to perform the aforementioned operations by exposing a set of fluent APIs. Listing 11 shows how it is possible to upsert (insert or update if already present) on the engine the rules defined and tested; it is sufficient to specify the identifier of the game and the list of the rules to ship to production. Thus, DSL4GaR allows a runtime adjustment of the rules to eventually adapt a game if it is not performing as expected.

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

Listing 11: Rule deployment on the gamification engine.

```
deploymentService.upsert(
    "62df8ddc0055b0275113d374", // id of the game in the
        gamification engine
    BadgeRules.getGreenBadge10Rule(),
    BadgeRules.getGreenBadge50Rule(),
    // Other rules
);
```

VI. Lessons Learnt

The case study highlights the advantages introduced by DSL4GaR. The lack of auto-completion mechanisms and syntax highlight is addressed by the IDE where the DSL4GaR is used into; the designers are constrained and guided during the whole rule definition phase, this drastically decreases typos and syntactical errors. Furthermore, for what concerns the simulation phase, through the graphical output provided by DSL4GaR during the simulation, the designers have the possibility to have a deep understanding of the actual consequences of the rules execution and their correlation. This allows to prevent the deployment of rules with unexpected behaviours and wrong mechanisms that might result in users dissatisfaction instead. Last but not least, even the deployment phase itself takes place inside the same environment. The designers can therefore handle the entire workflow of the rules without moving from the IDE they use and adapt the games at runtime based on the needs; this makes the entire process more robust and increases the overall flexibility.

VII. Conclusion and Future Works

DSL4GaR has introduced a series of advantages for the designers by providing an higher level of abstraction respect with the previous approach. It also opens the doors for a large set of possible evolutions and integration to further improve and empower the overall rules lifecycle.

A. Graphical abstraction layer

Game designers are typically non-programmers; on the one hand DSL4GaR reduces the gap among their domain knowledge and their coding skills by allowing the rule definition through Java (and only to specific subset of the language itself). On the other hand they are still required to write code

for designing rules. To further provide an higher abstraction level from the code, it could be introduced a graphical interface that leverages on the APIs of DSL4GaR to produce the final rules. In this way, the designers would be even more constrained on the possible values they could type/choose and thus to possible errors. Drools already provides an UI based software for rule definition that could be exploited and expanded to this purpose. Moreover, also MPS[8] could be introduced to pursue both an higher level of abstraction and to be exploited as Java DSL.

B. Java DSL for custom RHS generation

One of the main limitations of the DSL is related to the RHS generation; it currently produces Java code based on a set of pre-defined Velocity templates. It is not possible for the designers to define ad hoc logic for the consequences of the rules. To enable this feature, it would be necessary to integrate DSL4GaR with a Java DSL for the RHS definition; in this way the designers would be able to define arbitrary complex rules without being limited to the provided templates.

C. Gamification engine console

Another possible evolution that could be further analyzed consists in developing an extension of the gamification engine in order to provide a console that permits a direct interaction with DSL4GaR. This would allow to exploit DSL4GaR directly inside the engine in order design the next gamified systems.

REFERENCES

[1] S. Bassanelli, N. Vasta, A. Bucchiarone, and A. Marconi, "Gamification for behavior change: A scientometric review," *Acta Psychologica*, vol. 228, p. 103657, 2022.

[2] D. Johnson, S. Deterding, K.-A. Kuhn, A. Staneva, S. Stoyanov, and L. Hides, "Gamification for health and wellbeing:a systematic review of the literature," *Internet Interventions*, vol. 6, pp. 89–106, 2016.

[3] J. Hamari, "Gamification," in *The Blackwell Encyclopedia of Sociology*, G. Ritzer, Ed. John Wiley & Sons, Ltd, 2019, pp. 1–3. [Online]. Available: https://onlinelibrary.wiley.com/doi/10.1002/9781405165518.wbeos1321

[4] M. Sailer, J. U. Hense, S. K. Mayr, and H. Mandl, "How gamification motivates: An experimental study of the effects of specific game design elements on psychological need satisfaction," *Computers in Human Behavior*, vol. 69, pp. 371–380, 2017.

[5] B. Morschheuser, L. Hassan, K. Werder, and J. Hamari, "How to design gamification? a method for engineering gamified software," *Information and Software Technology*, vol. 95, pp. 219–237, 2018. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095058491730349X

[6] E. Loria and A. Marconi, "Reading between the lines – towards an algorithm exploiting in-game behaviors to learn preferences in gameful systems," in *International Conference on the Foundations of Digital Games*. ACM, 2020, pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3402942.3403016

[7] R. Orji, G. F. Tondello, and L. E. Nacke, "Personalizing persuasive strategies in gameful systems to gamification user types," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. ACM, 2018, pp. 1–14. [Online]. Available: https://dl.acm.org/doi/10.1145/3173574.3174009

[8] J. Vassileva, "Motivating participation in social computing applications: A user modeling perspective," *User Modeling and User-Adapted Interaction*, vol. 22, 2012.

[9] M. Proctor, "Drools: a rule engine for complex event processing," in *International symposium on applications of graph transformations with industrial relevance*. Springer, 2011, pp. 2–2.

[10] P. Browne and P. Johnson, *JBoss Drools business rules*. Packt Publishing Birmingham, 2009, vol. 1847196063.

[11] A. Bucchiarone, A. Cicchetti, and A. Marconi, "Towards engineering future gameful applications," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*. IEEE, 2020, pp. 105–108.

[12] A. Wasowski and T. Berger, *Domain-Specific Languages: Effective modeling, automation, and reuse*. Springer Nature, 2023.

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

[14] A. Bucchiarone, A. Cicchetti, F. Ciccozzi, and A. Pierantonio, Eds., *Domain-Specific Languages in Practice: with JetBrains MPS*. Springer, 2021.

[15] M. Völter and Itemis, "Best practices for dsls and model-driven development," *Journal of Object Technology*, vol. 8, no. 6, pp. 79–102, 09 2009.

[16] M. Fowler, *Domain-specific languages*. Pearson Education, 2010.

[17] F. Hermans, M. Pinzger, and A. v. Deursen, "Domain-specific languages in practice: A user study on the success factors," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2009, pp. 423–437.

[18] A. Van Deursen and P. Klint, "Domain-specific language design requires feature descriptions," *Journal of computing and information technology*, vol. 10, no. 1, pp. 1–17, 2002.

[19] P. Herzig, K. Jugel, C. Momm, M. Ameling, and A. Schill, "Gaml-a modeling language for gamification," in *2013 IEEE/ACM 6th International Conference on Utility and Cloud Computing*. IEEE, 2013, pp. 494–499.

[20] A. Bucchiarone, A. Cicchetti, and A. Marconi, "Gdf: A gamification design framework powered by model-driven engineering," in *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*. IEEE, 2019, pp. 753–758.

[21] A. Calderón, J. Boubeta-Puig, and M. Ruiz, "Medit4cep-gam: A model-driven approach for user-friendly gamification design, monitoring and code generation in cep-based systems," *Information and Software Technology*, vol. 95, pp. 238–264, 2018.

[22] R. Kazhamiakin, E. Loria, A. Marconi, and M. Scanagatta, "A gamification platform to analyze and influence citizens' daily transportation choices," *IEEE Trans. Intell. Transp. Syst.*, vol. 22, no. 4, pp. 2153–2167, 2021.

[23] W. Commons, "Fluent interface," 2022. [Online]. Available: https://en.wikipedia.org/wiki/Fluent_interface

[24] R. Kazhamiakin, E. Loria, A. Marconi, and M. Scanagatta, "A gamification platform to analyze and influence citizens' daily transportation choices," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, no. 4, pp. 2153–2167, 2021.

[25] R. Kazhamiakin, A. Marconi, M. Perillo, M. Pistore, G. Valetto, L. Piras, F. Avesani, and N. Perri, "Using gamification to incentivize sustainable urban mobility," in *2015 IEEE first international smart cities conference (ISC2)*. IEEE, 2015, pp. 1–6.

[26] M. Ferron, E. Loria, A. Marconi, and P. Massa, "Play&go, an urban game promoting behaviour change for sustainable mobility," *Interaction Des. Archit. J*, vol. 40, pp. 24–25, 2019.

[27] R. N. Landers and A. K. Landers, "An empirical test of the theory of gamified learning: The effect of leaderboards on time-on-task and academic performance," *Simulation & Gaming*, vol. 45, no. 6, pp. 769–785, 2014.

[28] K. Werbach and D. Hunter, *The gamification toolkit: dynamics, mechanics, and components for the win*. University of Pennsylvania Press, 2015.

[29] M. Csikszentmihalhi, *Finding flow: The psychology of engagement with everyday life*. Hachette UK, 2020.

[30] M. Hendrikx, S. Meijer, J. Van Der Velden, and A. Iosup, "Procedural content generation for games: A survey," *ACM Transactions on Multimedia Computing, Communications, and Applications (TOMM)*, vol. 9, no. 1, pp. 1–22, 2013.

[8] https://www.jetbrains.com/mps/