Modularizing and Evolving Applications using Scripting Modeling Languages

Michaël Hoste, Tom Mens
Service de Génie Logiciel
Université de Mons - UMONS
Place du Parc 20, 7000 Mons, Belgique
michael.hoste@umons.ac.be — tom.mens@umons.ac.be

Abstract
Domain-Independent Models are mainly used for documentation purposes and are most of the time too generic to be directly executed, even by code generation. Domain-Specific Models can sometimes be executed, but their scope is too specific to be reused for other purposes. We have developed a mechanism that allows the creation of modeling languages that will be directly executed into software applications. We were inspired by dynamic languages, especially scripting languages, and adapted their approach to models in order to be able to execute models directly, not for an entire application, but for a specific and well-defined part of it. The goal of scripting languages is to raise the level of abstraction of the host language and to delegate some work to an external language. With the help of two concrete examples, we illustrate that scripting modeling languages can meet this objective better than textual scripting languages and that an application can evolve only by using script models.

Keywords: visual scripting, model-driven software development, scripting language, modeling language, software evolution, model evolution

1. Introduction

One of the main advantages of dynamic languages is that they offer a higher degree of freedom than static languages. The developer does not need to declare variables, to allocate memory, or even to compile. The data types are also easier to manipulate. These features make dynamic languages good candidates for a junior developer to program [1].

Dynamic languages are also very appreciated as scripting languages. A scripting language is a mechanism that allows to control part of an application developed in another language. More and more applications use external scripting languages to extend their features (e.g., Adobe Photoshop, Blender, MySQL, VLC Media Player, Gimp, World of Warcraft and most recent games). Scripting languages can be helpful for internal development but, most of the time, their use is dedicated to external inexperienced developers wanting to make small customizations to their software. After reading a few tutorials, users with a low level of programming skills can create a new filter for Gimp [2], a new screen helper for World of Warcraft, or even a new campaign for a game if the game’s architecture allows it [3]. All low-level pieces of code are hidden behind an API (Application Programming Interface) created in the main language and accessible by the scripting language.

Knowing that dynamic languages, and more particularly scripting languages, seem perfect for beginners, the purpose of this article is to raise the level of abstraction to allow unexperienced developers to create executable models. Scripting modeling languages are models that are going to be executed in real-time in software using the underlying technology of scripting programming languages like Lua, Python or Ruby. Scripting modeling languages are intended to be transformed to scripting languages in real-time. The idea is to create scripting modeling languages that are easy to learn and easy to modify, yet with the same flexibility as dynamic languages [4]. For some domain-specific applications, we demonstrate that the application’s evolution can be managed only by model scripting, without any change to the main language.

Nowadays software engineers use more and more models as primary artifacts in the development of software systems. Model-driven software engineering (MDE) [5] addresses the intrinsic complexity of software-intensive
systems by raising the level of abstraction and hiding the accidental complexity of the underlying technology as much as possible [6]. Our approach could be interesting in the scope of MDE. UML is one of the most popular domain-independent software modeling languages [7]. UML 2.x provides 13 diagram types that correspond to as many different views on the software being modeled. As these views overlap on many points, it is very difficult to prevent the occurrence of inconsistencies between the views and thus maintain consistency between the different views during evolution [8] [9]. Because of that, only a few views are usually created to design an application and UML is mostly used for documentation purposes.

The goal of our paper is to include models in applications in a more integrated way. We integrate executable models as small parts of the software, as opposed to execute a model representing the entire software. In the traditional approach, a model represents a portion of an application architecture and design at a high level, and the programmer needs to refine the model with a textual programming language. Code generation is sometimes used but the application must be compiled again and some co-evolution issues between models and code may occur. In our approach, the model does not represent a first step that needs to be refined and reworked. The main idea is to delegate specific tasks to an external model the way we do with scripting languages, with real-time execution and avoiding co-evolution issues.

These days, more and more software applications use external scripting languages to extend their features. Section 2 presents the related work. Section 3 briefly summarizes the concept and the advantages of using both scripting programming languages and scripting modeling languages. Section 4 describes our vision of a scripting modeling language and the technical aspects of our framework. Section 5 validates, with the help of two examples, the advantages of using a model that controls the flow of an application. Section 6 highlights the multiple possibilities of evolving script models. Finally, section 7 presents future work and section 8 summarizes and concludes our work.

2. Related work

There is a lot of related work in the field of this article. Since a big part of our work is based on the transformation of a script model (defined in a scripting modeling language) to a textual script (defined in a scripting programming language), we are interested in the whole domain of code generation and model-to-text transformation from domain-specific modeling languages.

AToM3 is a tool that relies on graph rewriting techniques to perform tasks such as model transformation, code generation, operational semantics specification and visual modeling [10, 11]. One application of AToM3 in the domain of game character behavior has shown that it is possible to create an AI of a tank with a statechart and then generate the associated C++ code [12].

ATL [13, 14], QVT [15] and VIATRA2 [16] are model to model transformation tools. QVT is a standard created by the Object Management Group (OMG) that also defined the UML standard. ATL and VIATRA2 are using the Eclipse Modeling Framework (EMF)\(^1\). Some other tools use EMF to transform models by graph transformation [17].

Acceleo\(^2\) and Xpand\(^3\) are two EMF-based model-to-text generation tools. They are used to transform models into code through predefined or customized templates. Poseidon for DSLs\(^4\) is another EMF-based modeling tool that is used to create new DSLs and to generate code from them.

Another research domain directly related to our work is model execution. Java Application Building Center (jABC) [18] is a Java framework for model-driven application development that composes an application using executable building-blocks. It allows the creation of complex software systems by non-programmers.

A final related research domain is model-driven software evolution. Models, like other software artifacts, evolve during the software life-time [19, 20]. This evolution must be correctly handled to prevent the occurrence of inconsistencies in a models or between several models [8, 21, 22]. Furthermore, the models need to co-evolve with their metamodels [23].

---

1http://www.eclipse.org/modeling/emf/
2http://www.acceleo.org
3http://wiki.eclipse.org/Xpand
4http://www.gentleware.com/
3. Scripting languages

3.1. Scripting Programming Languages

Scripting programming languages (such as Python, Ruby or Lua) are often embedded in applications to extend their features. They are usually adopted for large applications where there is a need for modularity and where it is beneficial to delegate some tasks to an external process. Scripting languages have many advantages: (1) the scope of a script is limited to a subset of attributes, methods and classes through the use of a well-defined API; (2) a script can be modified and executed without any recompilation of the main program; (3) scripting languages provide, most of the time, a better productivity due to a simpler syntax provided by dynamic languages [1]; (4) scripting languages usually have the same capabilities as regular languages.

These advantages allow senior programmers to delegate some of the tasks to junior programmers that do not need to know the entire application architecture and code in order to be productive. Due to the limited scope of the script, there is a lower risk of side-effects and the propagation of errors is restricted to the script. As such, unexperienced users or programmers have the possibility to modify the application simultaneously with limited risk of breaking it. Furthermore, there is no need to install a complete tool-chain in order to compile a new version of the application. As scripts are dynamically interpreted, one can make some changes in the script and see the impact on the application behavior in real-time, without the need for recompilation.

Scripting programming languages are an easy way to involve the customer in the software development process. This allows for better feedback and faster adaptation to changing requirements. Smaller product life-cycles enable a more effective implementation that respond better to the user’s requirements [24].

There are many success stories involving scripting programming languages in both open-source and commercial communities. One out of many open-source examples is Blender, a powerful 3D modeling and animation application written in C/C++. Blender uses Python as scripting language to embed some behavior and provides an API with the available methods. By using Python scripting, the end-user can call Blender routines and extend the features in a wide range of ways without any need to recompile. There are now hundreds of scripts written for Blender that can be used by the community.

Some well-known commercial proprietary software also use scripts. Scripting is an easy way to allow end-users to extend the software with new features even when it is closed-source. With an active software community, it enhances the value of the software at virtually no cost. One very popular example of this is World of Warcraft, which uses a scripting language (Lua) to allow the player to modify and evolve the user interface in many ways. It is then possible to draw helpers on the screen, add a new world map, and log some 3D world information that appears on the screen.

3.2. Scripting Modeling Languages

Some studies have shown that using appropriate visual models is advantageous for a better understanding and a faster modification of software [25] [26]. However, it is unclear how hard it is to create a software design model that could define every aspect of a software application.

We propose to merge the advantages of both modeling and scripting languages. We have seen that scripting programming languages improve the software modularity. We claim that it is useful and possible to keep the modularity properties of scripting programming languages and raise the level of abstraction to the level of scripting modeling languages. Instead of including a textual scripting programming language in an application, we want to use a visual scripting modeling language with the same role.

The scripts will be more understandable, the development will be faster, and, consequently, its cost will be reduced. More importantly, part of the development can be outsourced to third-parties, non-professional programmers or even end-users, and all of these script models can be shared by the community. This kind of business model is a win-win situation for both the software provider and its clients.

For MDE, most of the time the software architect creates models and then the programmer implements them. That involves more work. Some solutions exist to automatically generate code from models or even to run models, but they still seem difficult to implement and still lack the proper tools [27]. In our solution, the model is considered as a primary artifact of the software and is directly executed.

---

5 http://wiki.blender.org/index.php/Extensions:2.4/Py/Scripts
6 http://www.wowwiki.com/Lua
4. Technical solution

We propose a framework to load and execute script models [28]. Our framework has the possibility to use and combine different scripting modeling languages and programming languages. Once a script model is created using a given scripting modeling language, it is transformed on-the-fly to a textual scripting language. Currently we support Lua, Ruby and Python. Our framework supports domain-independent modeling languages, like flowcharts (cf. Section 5.2), and domain-specific modeling languages (cf. Section 5.1). Software designers will have the possibility to create new scripting modeling languages adapted to their domain and needs.

We illustrate the technical aspects of our framework in Fig. 1. The left part of the figure represents the script model that conforms to the scripting modeling language. Different modeling languages can be supported to represent different types of models like flowcharts, state diagrams, or domain-specific models. Each time the script model evolves, a corresponding script, which conforms to a scripting language (Lua, Python, ...), is generated. This script is used by the main program through a library. This library is developed in the main programming language with the purpose to load and execute script models.

In practice, the library focus on two distinct parts: (1) the transformation of script models to textual scripts; (2) the smoothest possible integration of this process into a main programming language. These parts will be explained in the next sections.

4.1. Mapping Engine

The mapping engine takes care of the generation of a textual script from a script model. It is flexible enough to handle a wide range of modeling languages (including domain-specific ones) and any textual scripting language.

A possible solution would have been to create mappings between each scripting modeling language and each scripting programming language. We chose for a better solution consisting of going through an intermediate representation to make a bridge (cf. Fig. 2). This kind of solution also used in [29] has the advantage that the mappings
between the intermediate representation and every scripting programming language needs to be made only once and then reused. Similarly, for each new scripting modeling language a mapping needs to be created to the intermediate representation.

If \( m \) is the number of supported scripting modeling languages and \( n \) is the number of supported scripting programming languages, using the intermediate representation reduces the number of necessary mappings from \( m \times n \) to \( m + n \).

To make the mappings between models and code, we created an ad hoc solution that consists of parsing script models and using the intermediate representation to generate the code with the correct semantics. The whole process is explained on the following subsections. Note that the creation of a new mapping is a way to extend our framework. This is a task for a developer with good knowledge of the main language. Once a modeling language is taken into account in the framework, users can create and execute their script models and ignore the underlying technology.

4.1.1. Parser

The parser is the part of our library that loads a model, represented in XML format, and retrieves all useful information from it. An internal representation of the model is created and stored in memory. Our parser is based on JDOM, a very powerful XML processor for Java [30].

To create a parser for a new modeling language in our framework, we just have to inherit and specialize one general parsing superclass. This superclass contains everything that is required to make the parsing of XML models easier. The internal representation will be different depending on the modeling language.

Because our parser works at the level of XML files, it does not depend on a particular modeling syntax nor a specific modeling tool. But it is an advantage to use models created with the Eclipse Modeling Framework (EMF) [31] because we can parse them using existing EMF tools.

At this point, we only have the structure of our model in memory with no idea of its associated semantics. The next step is to use this structure and add some semantics to generate a textual script with the help of the intermediate representation.

4.1.2. Generator

The generator browses the internal representation of the script model obtained by the parser and add some semantics to it using the intermediate representation (cf. Section 4.1.3).

When creating a new generator, the developer does not mind what scripting language he wants to use. He just has to map the structure of the model to the intermediate representation. It is the role of the user that will use the library to choose what scripting language he wants to use and the intermediate representation will generate the right textual script in real-time.

One of the advantages of splitting the parser and the generator into two separate processes is that with the same model, we can use different semantics only by modifying the generator process and reusing the same parser process.

4.1.3. Intermediate Representation

The intermediate representation avoids the need to create a direct mapping from all modeling languages to all scripting languages. The representation consists of a set of methods that are called by the generator and that correspond to basic operations in every programming languages: loops, conditions, method calls, new objects, assignments, etc. When one of these methods is called by the generator, a line is added in a script file corresponding to a scripting language and the chosen textual script is generated in real-time.

In Code Fragment 1, we see the call of a basic method of the intermediate representation when creating a specific generator. Through the scripting language, this call will create a new Java object called artwork01 (the id depends on the internal representation of the model) that will be an instance of the class Artwork. Two parameters are given, the first one is a string and the second one is an object. The corresponding generated script lines in Lua, Ruby and Python are also written in Code Fragment 1.

The basic methods are listed in Code Fragment 2. We limited the number of operations to the ones we needed for our proof-of-concept case study (cf. Section 5). A lot of situations can already be managed by these simple operations but new kinds of operations will be added in the future to better handle some cases.

Working with this set of methods allows us to directly generate the right textual script without passing through another intermediate language. The script is created by a direct model to text transformation instead of having a
The specialization of the method `newObject()` for Lua, Ruby and Python

```java
// One basic method of the intermediate representation
sl.newObject("artwork"+id, "mobileMuseumGuide.data.Artwork", "String", "Number "+id,
   "Object", "position"+id);

// Corresponding generated script line in Lua
artworkO1 = luajava.newInstance("mobileMuseumGuide.data.Artwork", "Number O1", positionO1)

// Corresponding generated script lines in Ruby
include_class 'mobileMuseumGuide.data.Artwork'
artworkO1 = Artwork.new("Number O1", positionO1)

// Corresponding generated script line in Python
from mobileMuseumGuide.data import Artwork
artworkO1 = Artwork("Number O1", positionO1)
```

The set of methods that is used for the intermediate representation

```java
public abstract void print(String message);

public abstract void beginFunction(String name, String ... args);
public abstract void endFunction();

public abstract void beginCondition(Condition condition);
public abstract void nextCondition(Condition condition);
public abstract void endCondition();

public abstract void beginLoop(Condition condition);
public abstract void endLoop();

public abstract void newObject(String objectName, String className, String ... args);
public abstract void callMethod(String objectName, String methodName, String ... args);

public abstract void assignment(String variable, Expression expression);
```

To be able to transform the intermediate representation to a new scripting language, the abstract superclass with all the basic methods needs to be extended and specialized for the generation of the new scripting language. For example, if `beginCondition(Condition condition)` is called and needs to be transformed in Lua, "if "+condition.getCondition()+" then" will be written in the script, when keeping in mind the new indentation introduced by the condition (in Python, the indentation is part of the syntax).

Thanks to the intermediate representation, the generation of a scripting language needs to be implemented only once. Currently Lua, Python and Ruby are (at least partially) already taken into account and the framework can be extended easily.
4.2. Integration

The integration is the part of our library that allows a user to load a script model into the main programming language. It must allow developers to create dedicated APIs for script models. It is also responsible to load script models in a transparent way without even noticing the user that a textual scripting language is hidden behind the process. The user of the library is not supposed to modify or even see the generated script, he must always go through the script model.

Currently, our framework is only created for Java. It means that, at this time, we can only load script models into Java applications. But this library will be ported in the future to other main programming languages. The only condition is that the main programming language supports embedding of scripting programming languages and this is very common.

For Java, our library is built on top of other libraries that allow us to load Lua, Ruby and Python scripts. These libraries are respectively LuaJava [32], Jython [33] and JRuby [34]. The same kind of libraries also exists for other languages like C.

For the user of our framework, only a few lines are needed to load a script model:

- `ScriptingLanguage sl = new SLua();` initialize a scripting programming language (MyScriptingLanguage must extends ScriptingLanguage).
- `ScriptModel sm = new ScriptModel("path/to/model.dat", ml, sl);` load a model that conforms to MyModelingLanguage and that needs to be transformed to MyScriptingLanguage.
- `sm.execute(arg1, arg2, ...);` execute the script model with a list of parameters that will be given to the script.

4.2.1. The API

There is, to our knowledge, no way to formally restrict the scripting language to a part of the Java classes. Because (or thanks to) the Java Virtual Machine, each Java class can be accessed by the scripting programming languages. So, the accessible Java API corresponds to the whole set of Java classes. However, in order to pass an attribute or an object to a script model, it must be given as argument to the ScriptModel object at the time of execution.

However, when our framework will support other main programming languages that do not use a virtual machine, like C or C++, an API will be used in order to define the functions, methods and classes that will be accessible to the script models.

5. Validation

We provide a practical validation of our approach using the case study of a mobile museum guide [35]. A mobile museum guide is a device that is given to visitors at the entrance of a museum and that gives them information about the artworks they are passing by during their visit.

Museums often trade or move artworks depending on the current collections. In order to reduce the maintenance of expert programmers when artworks are changed and to improve the modularity of such a guide, an appropriate solution is to use scripts. This way, with only some documentation, an inexperienced programmer is able to change information (location, description, images, etc.) about all artworks without any need to hard-code or even compile the main code. The modifications will be taken into account in real-time. A second advantage of scripts in this context is that the museum manager can use them to create visit schedules adapted to the visitors’ characteristics (wheelchair, child, visually impaired) and preferences.

We present why model scripting is better than textual scripting for this purpose using the two examples of visit scheduling and interactive artwork presentations. The first example will use a domain-specific modeling language and the second one a domain-independent modeling language.

For better readability and space considerations, the two examples presented in this section are very simple but more complex ones could be handled in this situation depending on the users’ requirements.
Figure 3: The museum plan with a superimposed visit schedule using our scripting modeling language. The visual model has been created in Dia, and the corresponding textual script is generated automatically from it.

5.1. Domain-Specific Script Model (visit scheduling)

According to the visitors’ preferences, the museum manager needs to define more than one visit schedule using the mobile museum guide in the museum. Some visitors are interested in only one type of art, some others want to see the most important artworks in a short period of time, and others yet want to take the whole day to see everything. To avoid the need to hard-code a visit for each type of visitor and to start again when the museum is restructured, a scripting modeling language is used, just like the one shown in Fig. 3.

The script model incorporates the museum plan and allows the manager to modify the schedule of the visit. The script model has three layers.

The first layer is the architecture of the museum with the position of rooms, walls etc. This is the plan that will be displayed on the mobile museum guide during the visit.

The second layer corresponds to the position of all the artworks in the museum and to the associated picture that will be displayed on the mobile museum guide. If the museum manager wants to add, delete or change the position or the picture of an artwork, he must do it in this layer.

The third layer defines a path through the museum. The arrows that are on the script model are not displayed on the mobile museum guide but are used to represent a particular visit of the museum (cf. Fig. 4). The visitor will be guided through the museum depending on the positions of the arrows. This way, a museum can be restructured and a new visit schedule defined without the need to write a single line of code.

Once the script model of the museum visit is finished (using Dia\textsuperscript{7} in this case, a diagram creation program with really simple XML files to read), it is loaded in the main language (Java in our case) by our library and the corresponding textual script is generated and executed on-the-fly. Using Lua as scripting language, the code used to load the script model is displayed in Code Fragment 3 and the generated code is displayed in Code Fragment 4. Note that the code is verbose and quite ugly but this does not matter since the code is not intended to be seen and modified by a developer. It is generated and interpreted in real-time.

Note that when loading the script model, the parameter newGallery is given to sm.execute(). This parameter is transmitted to the script so that it can manipulate it. The script is also manipulating Java classes but this is quite transparent thanks to the Java virtual machine. All the extra work to access these methods is made in the scripting language itself and no special treatment must be done in Java code.

This easy access to Java classes allows us to create a mapping between the scripting language and the programming language. For example, we could retrieve the artwork descriptions from a database using the Java methods especially created for this purpose.

\footnote{http://projects.gnome.org/dia/}
Figure 4: The main interface of the mobile museum guide, taking into account the museum plan and visit schedule of the model in Fig. 3. The big picture and the description correspond to the artwork in front of which the visitor is standing. The small picture represents the next artwork the visitor must go to depending on the arrows in the script model.

**Code Fragment 3** The code used to load the MuseumPlan script model, to transform it in Lua code and to execute it

```java
// Initialize MuseumPlan modeling language and Lua scripting language
ModelingLanguage ml = new MLMuseumPlan();
ScriptingLanguage sl = new SLLua();

// Initialize, Transform and Execute the script model
sm = new ScriptModel("data/scripts/museumPlan.dia", ml, sl);
sm.execute(newGallery);
```
function callFunction(gallery)
  -- Artwork 01 definition --
  positionO1 = luajava.newInstance("guide.core.GeoPosition", 197, 41)
  pictureO1 = luajava.newInstance("guide.data.Picture", "1.png")
  artworkO1 = luajava.newInstance("guide.data.Artwork", "Number O1", positionO1)

  -- Artwork 02 definition --
  positionO2 = luajava.newInstance("guide.core.GeoPosition", 344, 43)
  pictureO2 = luajava.newInstance("guide.data.Picture", "2.png")
  artworkO2 = luajava.newInstance("guide.data.Artwork", "Number O2", positionO2)

  (...)
  -- Add artworks to gallery --
  gallery:add(artworkO1)
  gallery:add(artworkO2)
  (...)

  -- Path01 definition --
  nextO1 = luajava.newInstance("guide.data.NextArtwork", artworkO2)
  artworkO1:addInformation(nextO1)
end

5.2. Domain-Independent Script Model (interactive artwork presentation)

In the mobile museum guide, each artwork needs a presentation that may contain text, image, video, and sound, that must be displayed on the screen with a certain order. All of these pieces of information cannot be on the screen at the same time so the museum manager must choose a display order and some transitions. To ensure that a developer does not need to integrate these presentations into the mobile museum guide, the manager can use a flowchart like the one in Fig. 5. He can define an artwork presentation using predefined methods (ask(), drawScreen(), playMusic(), playVideo() or nextArtwork()) in the flowchart. These methods are actually defined in the main programming language (Java) and are part of the API that will be called into the scripting language.

In our example of La Velata in Fig. 4, the mobile museum guide provides the user with an optional introduction to the Renaissance before displaying the description of La Velata. Then, a video is played with the possibility to watch it again. Finally, the visitor is guided to the following artwork according to the visit schedule defined in the previous example.

This flowchart is loaded in the main language by our library (like in the Code Fragment 3 but with another modeling language) each time a visitor is close to the Velata artwork. The textual script is generated and executed on-the-fly and the artwork presentation is shown to the visitor. Using Lua as scripting language, the generated code is displayed in Code Fragment 5.

There are several ways of transforming flowchart states and transitions to code. Using a list of conditions into a loop when keeping the current state in memory is only one of them. It is also possible to walk through a chained list. The only work we need to do in the framework to use one of these techniques is to create another generator (using the intermediate representation) when keeping the same script model parser.

6. Model Evolution

The purpose of our scripting modeling languages framework is to delegate some parts of an application to an external process. This external process is the script model itself and will allow unexperienced developers to define the behavior or the specified part of the application. The script model can then be manually evolved by users to indirectly evolve the whole application. But it is also possible to evolve script models by automatic mechanisms. We propose
function callFunction()
    currentState = "012"
    while 1 do
        if currentState == "012" then
            currentState = "00"
        end
        if currentState == "00" then
            cond = ask("do you want an introduction on the Renaissance ?")
            if cond then
                currentState = "02"
            else
                currentState = "04"
            end
        end
        if currentState == "02" then
            playMusic("renaissance.mp3")
            drawScreen("RENAISSANCE_DESCRIPTION")
            currentState = "04"
        end
        if currentState == "04" then
            drawScreen("VELATA_DESCRIPTION")
            currentState = "013"
        end
        if currentState == "013" then
            playVideo("velata.avi")
            currentState = "016"
        end
        if currentState == "016" then
            cond = ask("do you want to watch the video again ?")
            if cond then
                currentState = "013"
            else
                currentState = "023"
            end
        end
        if currentState == "023" then
            nextArtwork()
        end
    end
end
three methods to automatically evolve script models: model optimization, model refactoring, and context adaptation. These methods are described in the following sections.

6.1. Model Optimization

Instead of trying to optimize the application itself using the main programming language, it could sometimes be useful to optimize the script model that is loaded through our framework. It is a way to externalize the complexity of a problem.

In the example of our mobile museum guide, a manager could have the need to systematically minimize the distance of all the visit schedules in its museum. An application that works on the script model can then be developed to minimize the paths through the museum without even having to change a line to the main program code.

This model optimization could also be used to adapt the visit schedules to persons with disabilities so that stairs are avoided and elevators are used instead. With this solution, the manager will not have to create separate visit schedules for those persons, they will be generated by the model optimizer.

6.2. Model Refactoring

Another mechanism that automate the evolution of our script model is model refactoring [36, 37, 38, 39]. The refactoring is the process of changing a software system in such a way that it does not alter the external behaviour of the code (respectively the model), yet improves its internal structure [40, 41].

For example, a model refactoring could be applied to flowcharts. Statements that are never reached can be deleted and duplicate states could be restructured so that the script model can be smaller. Any scripting modeling language (be it domain-independent or domain-specific) could benefit from model refactoring [42].

6.3. Context Adaptation

Sometimes, models need to be adapted to a specific context [43]. For example, suppose that our scripting modeling language allows us to define the position and style of the graphical elements on the mobile museum guide interface. The script model could automatically be transformed in real-time depending on various factors such as the age of the visitor (funny fonts for the youngest, big fonts for the oldest), depending on the ambient luminosity (high or low contrast) and so on. This context adaptation could be especially useful for embedded software and mobile devices with lots of sensors (light, accelerometer, microphone, GPS, GSM, wifi, etc.).

7. Future work

In the future, we want to integrate script models in other use cases. We have the intuition that this approach could benefit to users that want to easily create decision trees with modeling languages similar to flowcharts. One other area that might be interesting to explore is the use of script models in the domain of artificial intelligence. We could imagine to control a robot in real-time by modifying the corresponding script model.
We also intend to create new domain-independent and domain-specific scripting modeling languages. Among others, we want to be able to execute UML state machines [44] or UML statecharts [45]. These modeling languages could represent a good solution to integrate some behavior into an application. Again, a good use case for these modeling languages is the control of a robot [12].

Instead of creating new use cases from scratch, we intend to validate our approach by extending existing open source applications with the use of script models. This way, we hope that the related community will provide us with feedback.

Another goal of our framework is to allow users to easily create script models based on new modeling languages that correspond to their needs. One way to integrate this feature in our framework is to rely on Poseidon for DSLs 8 or other similar initiatives. Poseidon for DSLs is a tool that allows the creation of a new domain-specific modeling language in only a few hours. This new DSL can then be manipulated with a modeling tool that is especially designed for this purpose. A great advantage of this approach is that the generated models are based on the Ecore metamodel and EMF provides a very powerful framework to deal with these models. The appearance of inconsistencies is very common in the domain of MDE [21, 22]. Using EMF technology will help us to ensure that there are no inconsistencies introduced in the script model during its creation.

A challenge for the future of our framework is to address the issue of the co-evolution of scripting modeling languages and script models, or even the co-evolution of API and script models that used this API. This is a problem that is often discussed in literature [36, 46, 23].

8. Conclusion

We have shown in this article that our framework is capable of raising the level of abstraction of textual scripting languages to obtain scripting modeling languages. It is able to execute models in real-time in software using the underlying technology of scripting programming languages. The main advantage of this is that less-experienced developers can easily change an application’s behavior or add a behavior at a high level of abstraction and without installing a complete tool-chain or even recompiling the application.

Script models can be directly integrated and executed as part of an application. Depending on the needs of the application, scripting modeling languages can be domain-specific or domain-independent. Our framework is flexible enough to be independent of the scripting modeling language, programming scripting language, API or even main programming language supposing the library is ported on other programming languages.

Through two concrete examples concerning a mobile museum guide, we have seen that, in the context of concrete problems, raising the level of abstraction improves the modularity of the application. This modularity improvement enables users to modify and evolve the application’s themselves just by modifying the script models and without bringing any change to the main programming of the application. This flexibility and the use of script models will make software evolution an easier and cheaper task.

The manual or automatic evolution of script models also provides us some interesting ways to automatically evolve an application or to adapt it to a specific context.

Acknowledgments

This research is funded by the Ministère de la Communauté Française - Direction générale de l’Enseignement non obligatoire et de la Recherche scientifique, Belgique (Action de Recherche Concertée AUWB-08/12-UMH19)

References


8http://www.gentleware.com/poseidon-for-dsls.html

